Scientific Programming with C++

MULTIPLE INHERITANCE, FRIENDSHIP, ABSTRACTION, ENCAPSULATION

Learning objectives

- After this lecture and related assignments, you will…
  - understand how multiple inheritance works
  - have an idea of how friendship works
  - be introduced to design philosophy of object-oriented programming

Inheritance

- We know that a class can derive members from a parent class
  - Methods
  - Attributes
- For example, a Car class can derive members from a more general Vehicle class
  - Methods: accelerate(), brake(), startEngine()
  - Attributes: numberOfTires, numberOfSeats, mass
- We also know that embers can be inherited across multiple “levels”
  - For example: Vehicle to Car to ElectricCar
- A single class can also inherit members from several parent classes
  - Multiple inheritance

Multiple inheritance

- A class inherits members from several other classes
- Order of inheritance matters
  - Constructors are called in inherited order
    - Derived class last
  - Destructors are called in reverse inherited order
    - Derived class first

Example: inheritance order (1/2)

```
class FlyingOrganism {
public:
    FlyingOrganism() {
        std::cout << "I am a flying organism." << std::endl;
    }
    ~FlyingOrganism() {
        std::cout << "Flying organism is being destructed." << std::endl;
    }
};

class TerrestrialAnimal {
public:
    TerrestrialAnimal() {
        std::cout << "I am a terrestrial animal." << std::endl;
    }
    ~TerrestrialAnimal() {
        std::cout << "Terrestrial animal is being destructed." << std::endl;
    }
};

class Butterfly : public TerrestrialAnimal, public FlyingOrganism {
public:
    Butterfly() {
        std::cout << "I represent a species of butterfly." << std::endl;
}
    ~Butterfly() {
        std::cout << "Butterfly is being destructed." << std::endl;
    }
};
```

Example: inheritance order (2/2)

```
class Butterfly : public TerrestrialAnimal, public FlyingOrganism
```

```
int main()
{
    // constructors will be called in order
    Butterfly monarch;
}
```

```
I am a terrestrial animal.
I am a flying organism.
I represent a species of butterfly.
Butterfly is being destructed.
Flying organism is being destructed.
Terrestrial animal is being destructed.
```

Multiple inheritance: ambiguity

- If the parent classes have identically named members, compiler won't know which one to use when the derived class accesses them
  - Compiler error: ambiguity
- Parent class can be specified with the scope resolution operator (::)
  - derivedObject.FirstParentClass::doStuff();
  - derivedObject.SecondParentClass::doStuff();

Example: ambiguity (1/4)

```
class FlyingOrganism {
public:
    FlyingOrganism() {
        std::cout << "I am a flying organism." << std::endl;
    }
    ~FlyingOrganism() {
        std::cout << "Flying organism is being destructed." << std::endl;
    }
    void fly(double distance) {
        std::cout << "Flying with my " << methodOfFlight << "  for " << distance << " meters!" << std::endl;
    }
    void rest(double duration) {
        std::cout << "Flying organism is resting for " << duration << " hours!" << std::endl;
    }
protected:
    std::string methodOfFlight = "unspecified";
};
```

Example: ambiguity (2/4)

```
class TerrestrialAnimal {
public:
    TerrestrialAnimal() {
        std::cout << "I am a terrestrial animal." << std::endl;
    }
    ~TerrestrialAnimal() {
        std::cout << "Terrestrial animal is being destructed." << std::endl;
    }
    void walk(double distance) {
        std::cout << "Walking with my " << numberOfLegs << " legs for " << distance << " meters!" << std::endl;
    }
    void rest(double duration) {
        std::cout << "Terrestrial animal is resting for " << duration << " hours!" << std::endl;
    }
protected:
    unsigned int numberOfLegs = 0;
};
```

Example: ambiguity (3/4)

```
class Butterfly : public TerrestrialAnimal, public FlyingOrganism {
public:
    Butterfly() {
        std::cout << "I represent a species of butterfly." << std::endl;
        numberOfLegs = 6;
        methodOfFlight = "flappy wings";
    }
    ~Butterfly() {
        std::cout << "Butterfly is being destructed." << std::endl;
    }
};
```

Example: ambiguity (4/4)

```
int main()
{
    // constructors will be called in order
    Butterfly monarch;
    monarch.walk(0.1); // inherited from TerrestrialAnimal
    monarch.fly(0.5); // inherited from FlyingOrganism
    //monarch.rest(1); // error: "Butterfly::rest" is ambiguous
    // fixed by specifying the parent class to get the function from using the scope resolution operator ::
    monarch.FlyingOrganism::rest(1);
    monarch.TerrestrialAnimal::rest(1);
}
```

```
I am a terrestrial animal.
I am a flying organism.
I represent a species of butterfly.
Walking with my 6 legs for 0.1 meters!
Flying with my flappy wings  for 0.5 meters!
Flying organism is resting for 1 hours!
Terrestrial animal is resting for 1 hours!
Butterfly is being destructed.
Flying organism is being destructed.
Terrestrial animal is being destructed.
```

When do you use multiple inheritance?

- When your class has an "is-a" relationship to its potential parent classes
  - Butterfly is a flying organism
  - Butterfly is a terrestrial animal (mostly in larval stage)
- When your class has an "has-a" relationship to its potential parent classes, you should use composition instead of multiple inheritance
  - Composition: an object is at least partially made up of other objects
  - Motorcycle has an engine (but is not an engine)
  - Motorcycle has a seat (but is not a seat)
  - Therefore, motorcycle shouldn't inherit Engine and Seat classes, but have them as its members
- Multiple inheritance is seldom necessary
  - Use can lead to the "diamond problem"
    - A class inheriting two classes that inherit the same base class, effectively duplicating it

Friendship

- Functions and classes can be declared "friends" of another class
  - Grants them access to protected and private members of the class
- Friendship is one-way
  - Even if A is a friend of B, B is not necessarily a friend of A ☹
    - Unless B is also declared as a friend of A
- Friendship is not inherited
- Useful when
  - Overloading operators belonging to class A that must have access to members of class B
  - You want an object to use restricted members of a class without that object inheriting the class
- Improves encapsulation

Example: function friendship

```
class LiquidSample {
    friend std::ostream& operator<<(std::ostream& os, const LiquidSample& sample);
public:
    LiquidSample(std::string label, double conc, double vol) {
        label_ = label;
        concentration_ = conc;
        volume_ = vol;
    }
    void dilute(double volume) {
        concentration_ = concentration_ * (volume_ / (volume_ + volume));
        volume_ += volume;
    }
private:
    std::string label_ = "UNLABELED"; double concentration_ = 0; double volume_ = 0;
};

std::ostream& operator<<(std::ostream& os, const LiquidSample& sample)
{
    os << sample.label_ << " concentration: " << sample.concentration_*100 << "%";
    return os;
}

int main() {
    LiquidSample HF("hydrofluoric acid", 1.0, 2.0);
    std::cout << HF << std::endl; HF.dilute(1); std::cout << HF << std::endl;
}
```

```
hydrofluoric acid concentration: 100%
hydrofluoric acid concentration: 66.6667%
```

Abstraction and encapsulation

- Concepts of designing C++ programs
- Closely tied to object-oriented programming (classes enable both)
- Changing internal implementation won't affect outside code and vice versa
- Abstraction considers ease of use; encapsulation protects information from outside use

Abstraction

- Data abstraction
  - Only the minimum required information is shown to the user
  - Unnecessary data is hidden
  - Example: access specifiers in classes (public, protected, private)
- Control abstraction
  - Only the minimum required implementation is shown to the user
  - Example: header files that only show function signatures
    - Allows outside users to know how to call them
    - Doesn't reveal implementation details

Encapsulation

- Information and data are wrapped within programmatical entities to protect them from public access
- Data protection
  - Internal state of object is protected because only pre-defined public methods can access it
    - Prevents access from any code outside the public methods
- Information hiding
  - Use of classes is more secure because only their public interface is available to an outside user
- Example: use of private variables and their modification with accessor functions (get and set) so that code outside the class cannot directly access them

Summary

- Classes can inherit from multiple parent classes
  - Order of inheritance is important
  - Ambiguities in member names must be resolved
- Friendship enables access to non-public members of a class
- Classes help achieve both abstraction and encapsulation
  - Abstraction hides unnecessary information and makes the user interface of code more approachable
  - Encapsulation protects information by defining where in code it can be accessed and how