Inheritance in C++

Peter Chapin

CIS-3012, C++ Programming

Vermont State University

Finish Me!

- *Talk about the basics…*

Multiple Inheritance (MI)

- Very controversial feature
  - Adds considerable complexity to the language
  - Is rarely needed
  - Regularly misused
- Critiques of C++ point to MI as a design error (not worth the trouble)
- Advocates of C++ talk about how MI lets you express designs that are very awkward any other way (i.e., in other languages)
- Probably both are right! ☺
- Most modern languages do not have MI, although some do

What is Multiple Inheritance?

- MI is the ability to inherit from multiple base classes at once
  - All the methods in all the bases are combined in the derived class
  - *All the data members in all the bases are part of the derived class**
  - A derived object can be passed to function expecting a reference to any of its bases.
- But Java…?
  - Java allows multiple interfaces to be implemented at once
  - This provides much of the same functionality, but it more limited. Often this is enough
  - MI is C++ is more general, providing all the abilities of Java's interfaces and more

* This is the one that cases the complexity

More On Interfaces

- C++ doesn’t have a distinct concept of “interface” the way Java does
- However, an interface can be defined as a class in which all methods are pure virtual (i.e., have no implementation or *abstract* as they are typically called)
- In many examples, such classes have no data members and thus look exactly like Java interfaces
    - However, C++ allows such classes to have data members if desired
- Then MI is used to “implement” multiple “interfaces.” This use of MI is not problematic, and is not what most people are thinking when they talk about MI

Basic Syntax of MI

```
class B1 { … };
class B2 { … };
class D : public B1, B2 { … };
```

Base class list can include an arbitrary number of base classes

We will only consider public inheritance.
Although private and protected inheritance also exists, they are rarely used
Inheritance in other OOP languages is always public
  (C++ is the only OOP language that offers multiple inheritance modes)

Is-A? Has-A?

- It is often said that inheritance is appropriate if the derived object is a base object ("is-a"):
  - A car is a vehicle. A truck is a vehicle.
  - Thus, class Car should be derived from class Vehicle, etc.
  - The methods of Vehicle apply to all vehicles, such as cars, although every specific Vehicle class may have its own way to do things.
- Composition (including one object inside another) is appropriate if the containing object has a contained object ("has-a"):
  - A car has a engine. A car has a set of wheels
  - Thus, class Car should have a member of type Engine, etc.

With Multiple Inheritance?

- MI says that the derived object is-a instance of each of its bases

```
class Mammal { ... };
class WingedAnimal { ... };

class Bat   : public Mammal, WingedAnimal { ... };
class Eagle : public Bird,   WingedAnimal { ... }
class Whale : public Mammal, MarineAnimal { ... }
```

- A Bat is a Mammal, but a Bat is also a WingedAnimal, and similarly for the other examples.

Continuing…

```
void make_fly( WingedAnimal &creature );
void clean_hair( Mammal &creature );

Bat vampy;  // So cute!

make_fly( vampy );   // OK. Vampy is a WingedAnimal and thus can fly
clean_hair( vampy ); // OK. Vampy is a Mammal and thus has hair
```

Inside the `make_fly` function:

```
void make_fly( WingedAnimal &creature )
{
    // …
    while( creature.flight_speed( ) < 100 )
        creature.flap_wings( );
    // …
}
```

Both `flight_speed` and `flap_wings` are methods of `WingedAnimal`

Class WingedAnimal

```
class WingedAnimal {
public:
    virtual int flight_speed( ) const;
    virtual void flap_wings( ) = 0;

private:
    int current_flight_speed;
};
```

Pure virtual, meaning that every derived class must override
Every `WingedAnimal` flaps its wings differently

Class Mammal

```
class Mammal {
public:
    virtual int hair_dirtiness( ) const;
    virtual void brush_hair( BrushType brush, int stroke_count ) = 0;

private:
    int hair_dirtiness_coefficient;
};
```

Both base classes have data members!

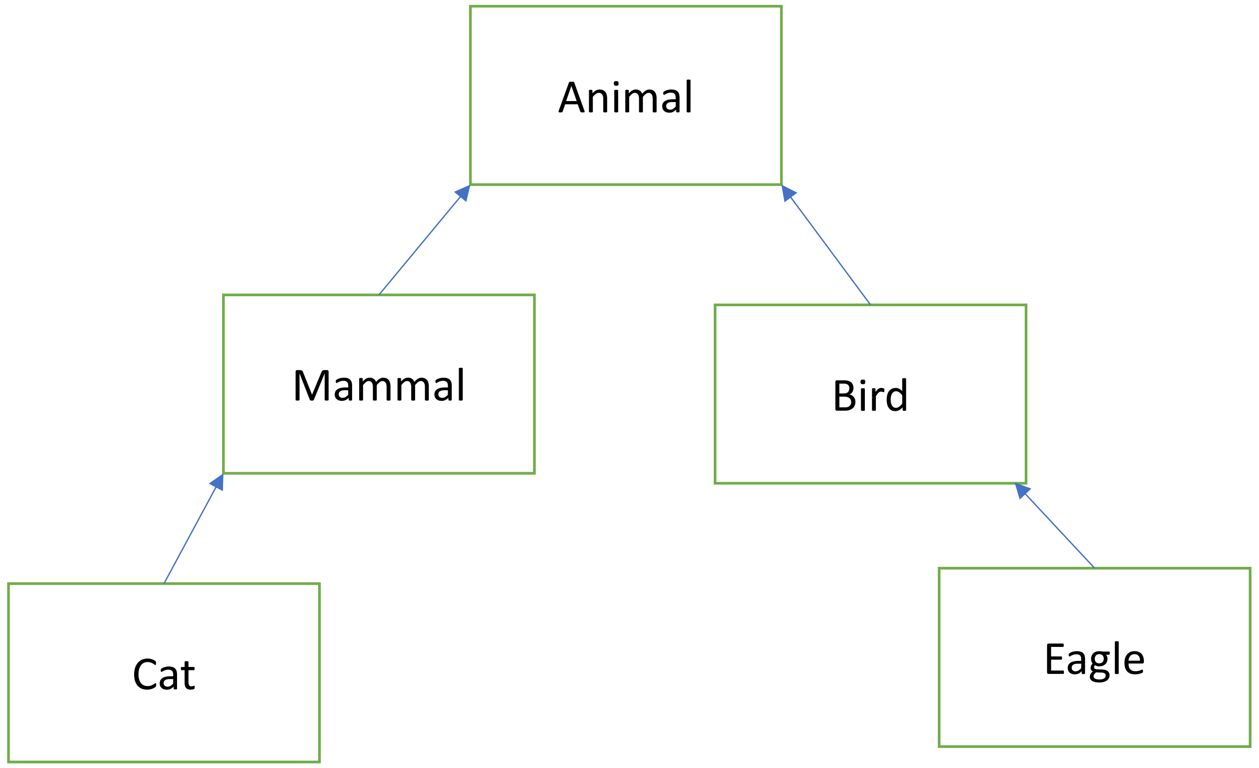
Class Bat

```
class Bat : public Mammal, WingedAnimal {
public:
    // How to brush a bat’s hair.
    void brush_hair( BrushType brush, int stroke_count ) override;

    // How a bat flaps its wings.
    void flap_wings( ) override;

    // Inherit flight_speed and hair_dirtiness from the base classes

private:
    // Specialized data members relevant only to bats.
    GPSCoordinate home_cave_location;
};
```

Since Bat overrides all pure virtuals from its bases, it is permitted to create a Bat instance

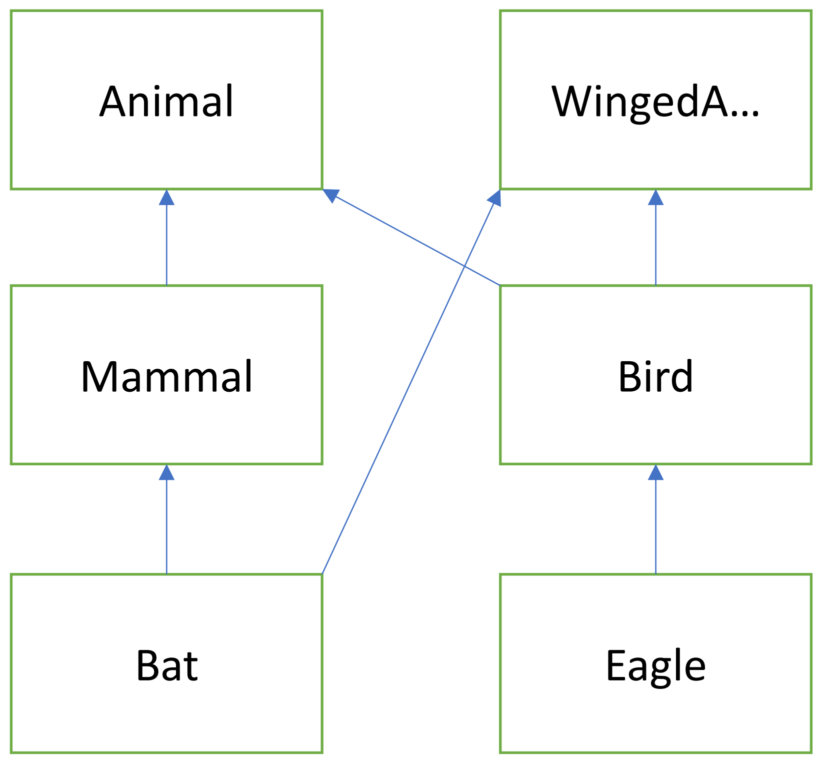
What’s the Controversy?

- Without MI, class diagrams are always trees:

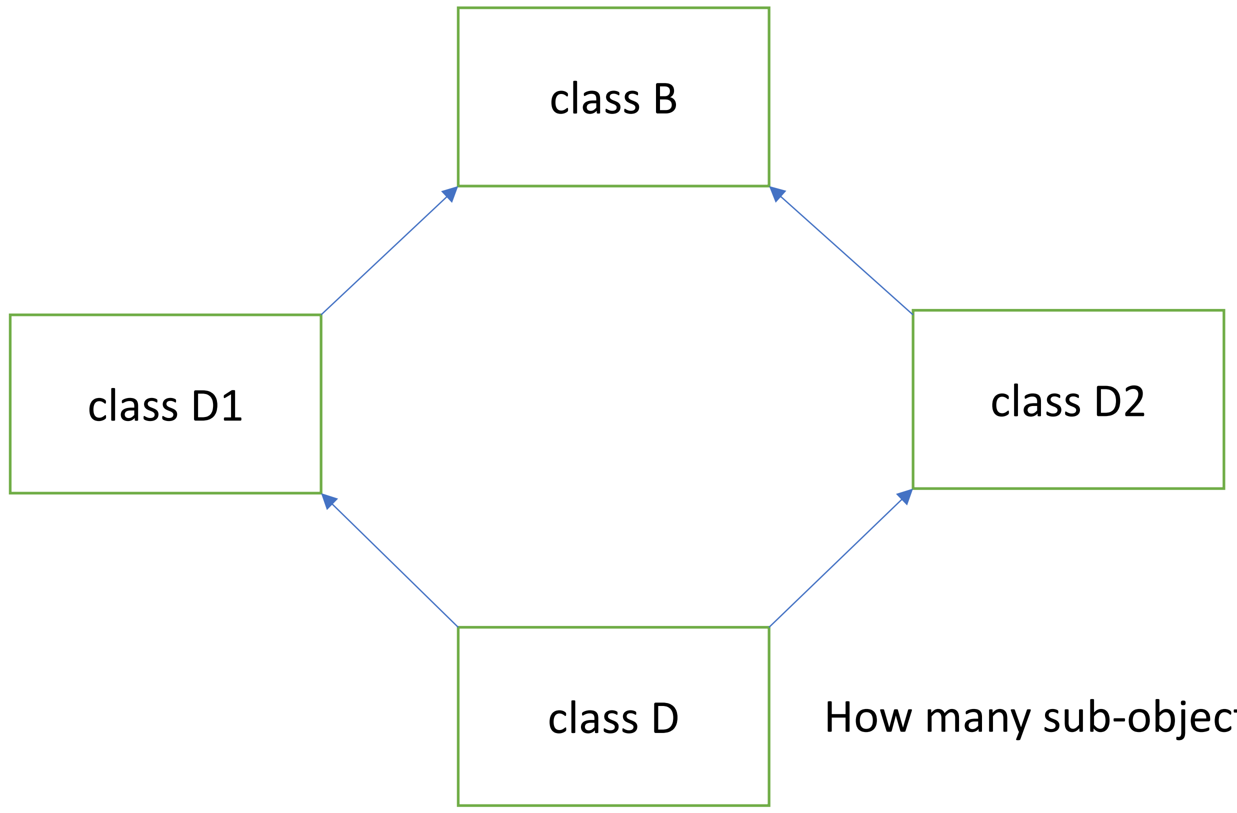


… But with Multiple Inheritance…

- Class diagrams are now DAGs:



The Diamond Inheritance Graph



How many sub-objects of type B are contained in a D?