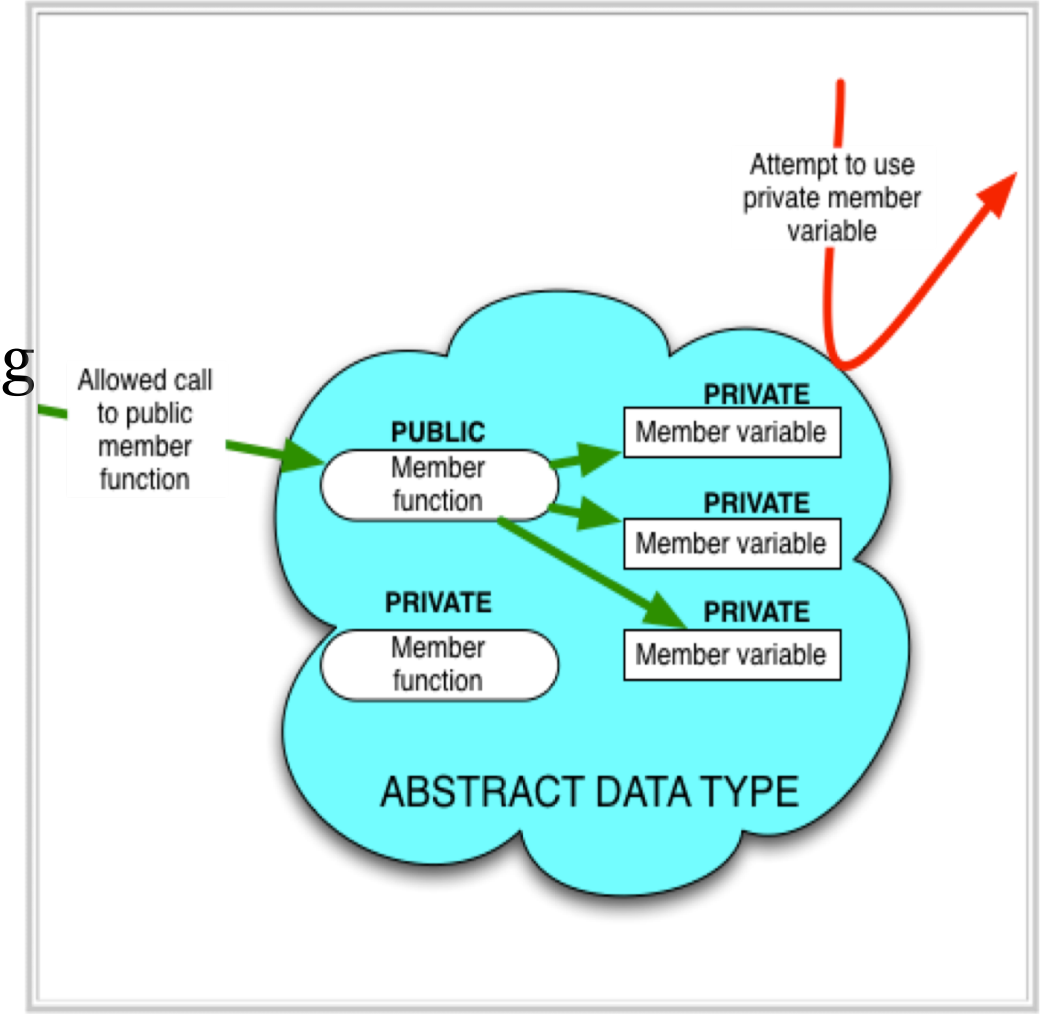
School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Class Hierarchies & Inheritance

Review

- ADT and Black boxes
- Interface
  - Public member functions
  - Description
- Interface is the only thing a user of ADT needs to know
  - Information hiding
  - Benefits?
- Three rules to make a class an ADT



Overview

- Class
- Objects = data + member functions.
- Design
- In this course we will frequently use classes that are very similar to each other, but not quite the same.
- Can we use elements of C++ to make this more efficient?

Design

- When you design software, you should design it in such a way that it:
  - Solves the problem correctly.
  - Does so efficiently.
  - Is easy to maintain
  - Is potentially applicable to other areas.

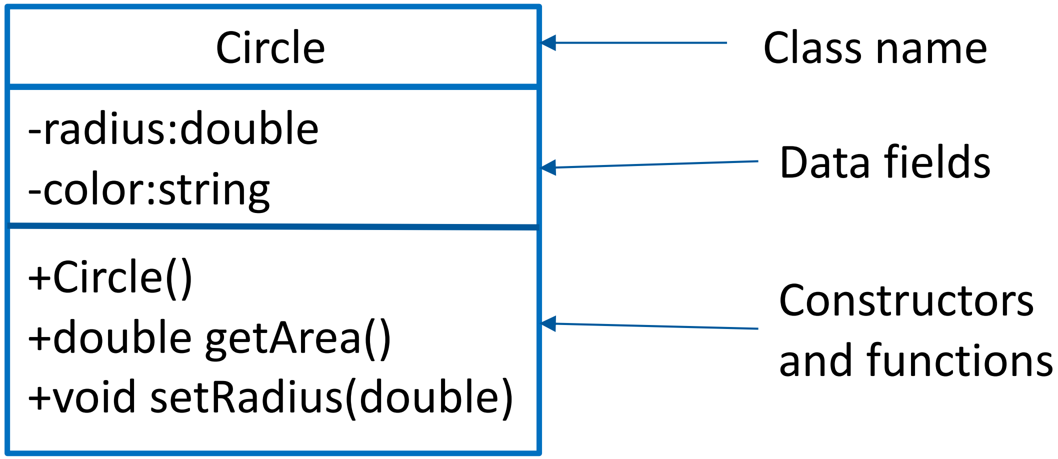
Design

- You are familiar with OOP

- Consider the entity "student"
  - information : name, address, field, scores, etc
  - Some functions, or behaviors: e.g. calculate GPA

Classes

- A class is a template or a blueprint that defines what an object's data and functions will be.
- Objects will call methods on each other.
- The methods should be strongly associated with the data of that class.
- Practice and experience -> Good design
  - distinct groupings of variables
  - separate parts of the desired behavior.



Circle

-radius:double
-color:string

+Circle()
+double getArea()
+void setRadius(double)

Separating Behavior

- Make sure that the separation:
  - makes sense
  - efficient

Example

- Consider a class of vehicles. There are many common features that all vehicles have.
  - Variables include:
    - carrying capacity, number of passengers
  - Methods include:
    - add passenger, move vehicle
- However, there are also some features that only belong to a certain type of vehicles.

Why hierarchies?

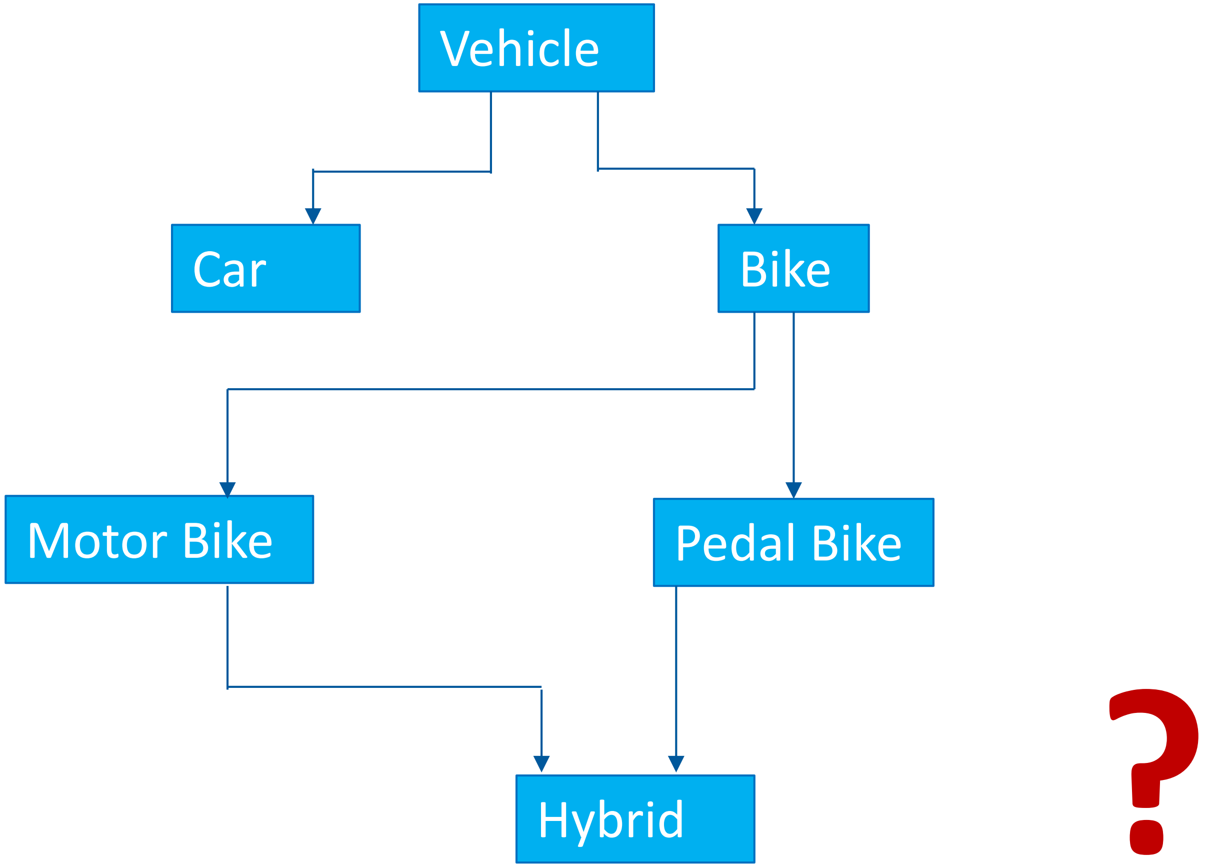
- We could write a Vehicle class, a Car class, a Bike class, an Aircraft class, which are all similar but not quite the same.
- Vast duplication of code and hard to maintain
- Why not take a different approach and build classes out of OTHER classes?

- The class hierarchy defines the inheritance relationship between the classes.

Example

- Consider the Vehicle class, and now consider two subclasses of Vehicle, Cars and Bikes.
- What're the differences between a car and a bike?
- How do we model these in terms of:
  - variables?
  - methods (behaviours)?

Example



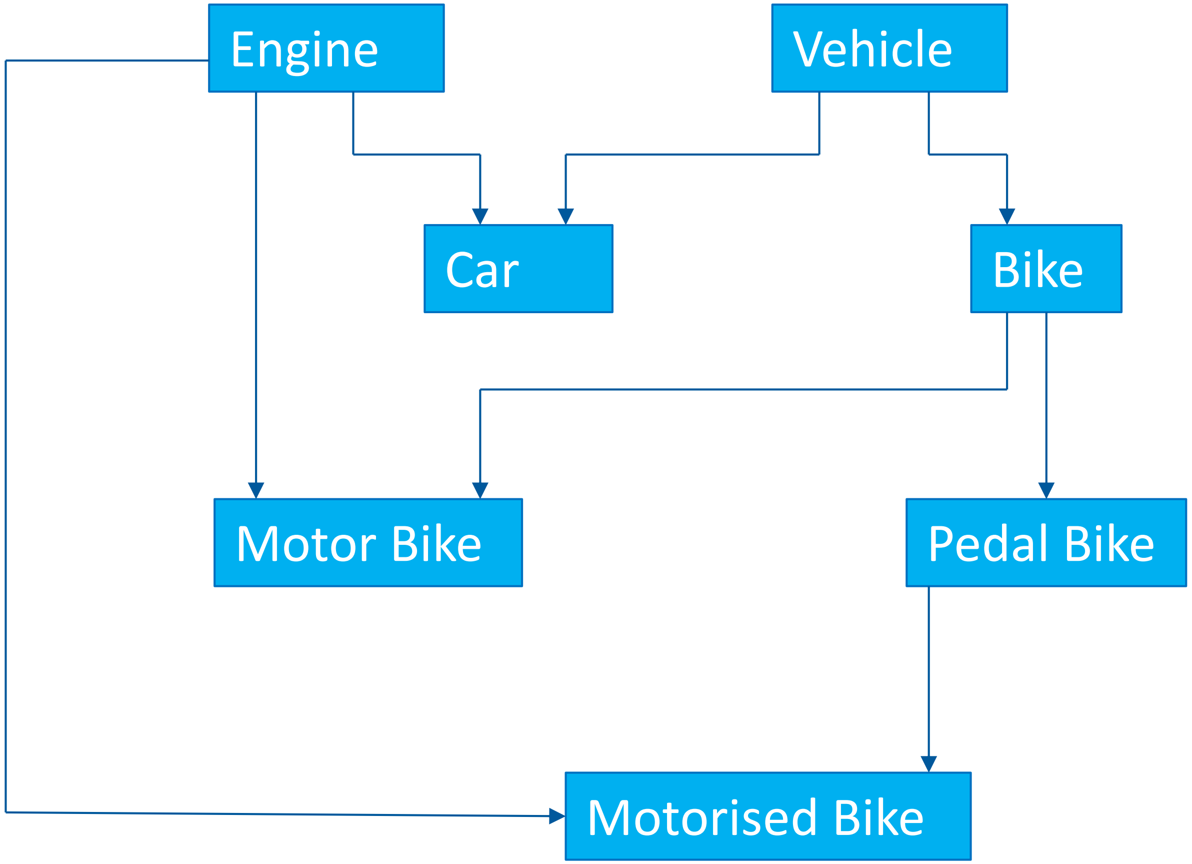
The classes are separated based on concepts.

Class Hierarchies

- One key problem in constructing a class hierarchy:
  - focus on the relationships between the concepts without considering the behaviours and how you'd better implement them.
- Is a Car a separate class near the top or is it just a vehicle with an engine with four wheels?
- Are we adding a behaviour or changing a default?

- We should design the class hierarchy based on our requirements.

Example- separating by behaviours



What impact does multiple inheritance have on the methods that we choose?
How do we handle this in C++?

Class Hierarchies

- The class hierarchies that you should form as part of your design must:
  - solve the problem in hand
  - be efficient
  - reduce code duplication
  - be well-defined and clearly understood
- Class hierarchies allow us to build classes in a way that we can build on existing classes to make new ones.
  - Get it right once, then we can re-use it.
  - But how do we re-use it?

Inheritance

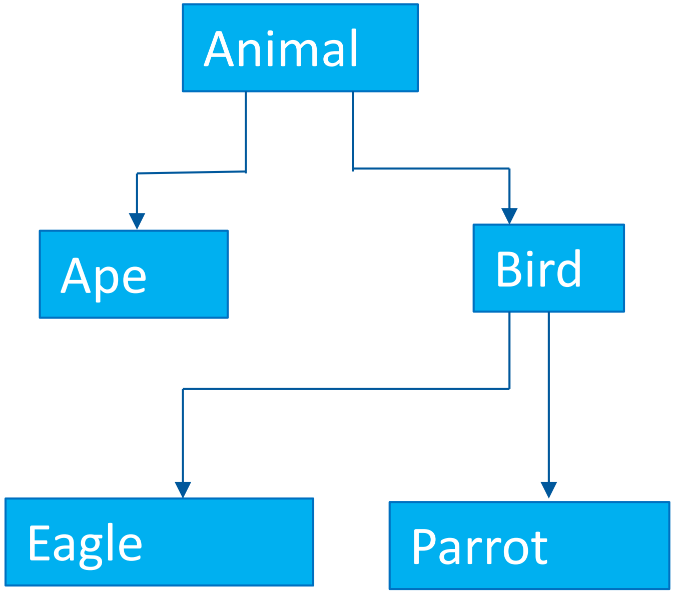
- Object-oriented programming allows you to derive new classes from existing classes. This is called inheritance.

- A class ***A* extended** from another class *B* is called a **derived class**. *B* is called a **base class**. Class *B* is also called a **parent class** while class *A* is the **child class**.

- A derived class and its base class must have the ***is-a*** relationship.

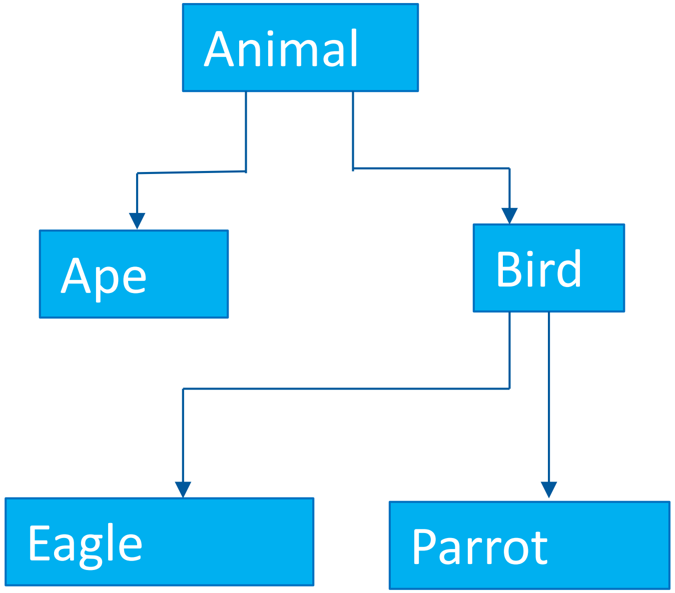
Example 1

- Consider the example hierarchy starting from Animal, with subclasses Birds and Apes.
- Why do we have Birds, then Eagles and Parrots?
  - Birds are Animals.
  - Animals may have heart rates, geographical location and food type.
  - Birds have wing spans, feather type and egg colours.
- Every Bird is an Animal but not every Animal is a Bird!

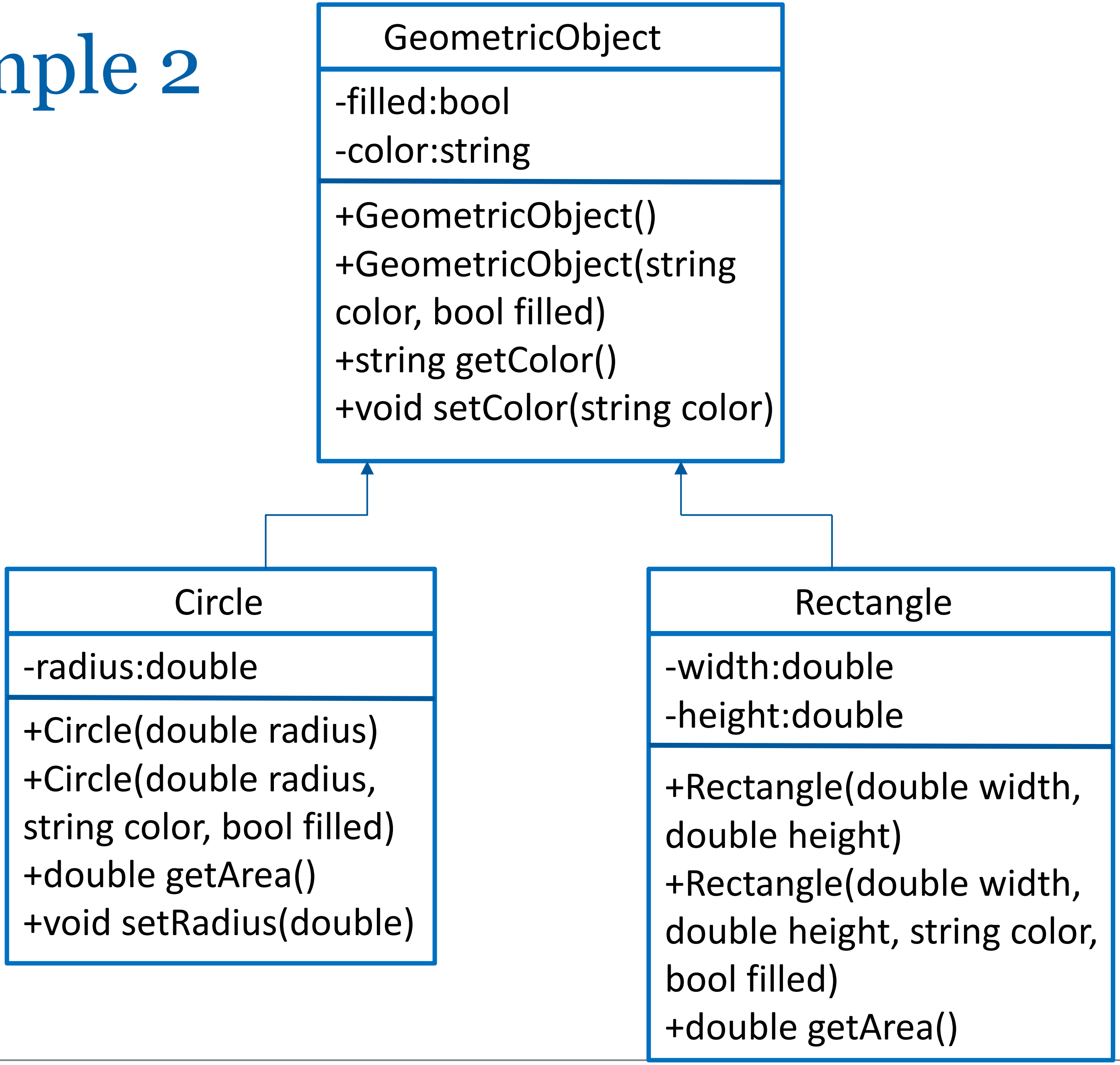


Family relationships

- The derived class Bird is a **subclass/child class** of Animal.
- Animal is a **parent class** of Bird.
- Bird *inherits* the member functions of Animal.



Example 2



GeometricObject

-filled:bool
-color:string

+GeometricObject()
+GeometricObject(string color, bool filled)
+string getColor()
+void setColor(string color)

Circle

-radius:double

+Circle(double radius)
+Circle(double radius, string color, bool filled)
+double getArea()
+void setRadius(double)

Rectangle

-width:double
-height:double

+Rectangle(double width, double height)
+Rectangle(double width, double height, string color, bool filled)
+double getArea()

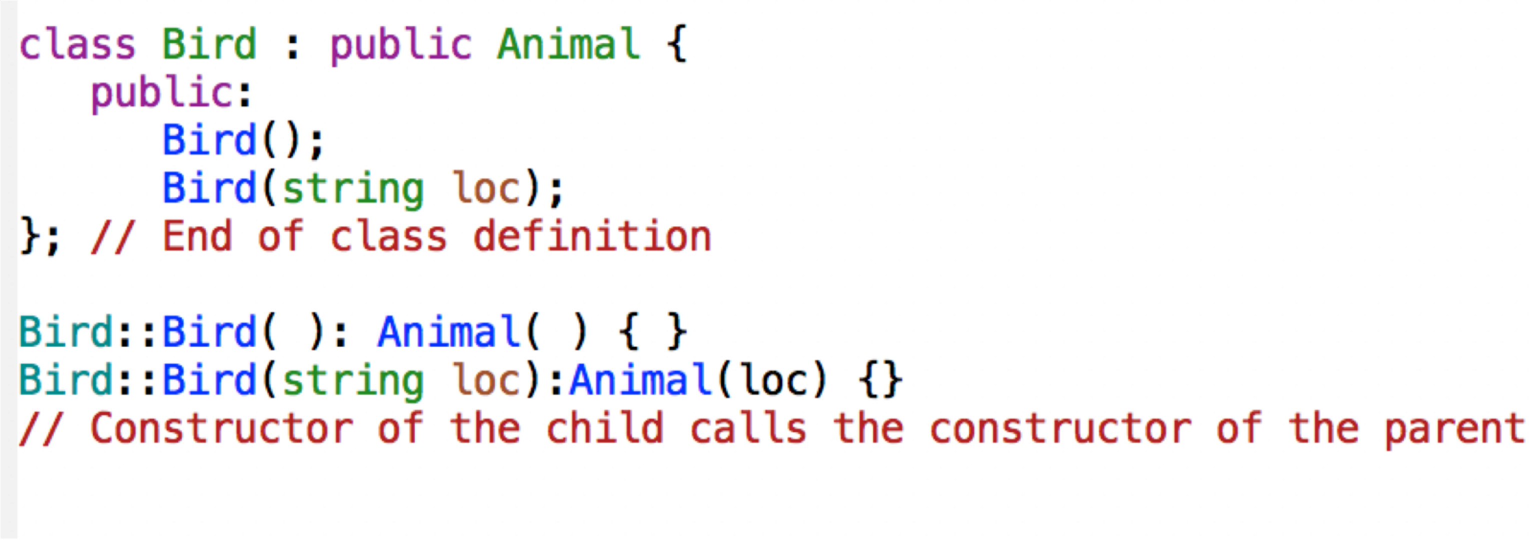
Child Class

- The child class inherits, by default, all public member variables and all public member functions of the parent.
- We can add member functions and variables to the child, these are not available to the parent.
  - These are available to children of the child!
- We can also redefine/override the member functions of the parent as viewed by the child.

Advantages

- If the child classes inherit methods from the parent and then we change the parent, all children automatically get access to the new, improved code.
- We can use the child in places where we could use the parent, although we are restricted to the methods publicly available from the parent.
- Very, very powerful technique.

Example



```
class Bird : public Animal {
   public:
      Bird();
      Bird(string loc);
}; // End of class definition

Bird::Bird( ): Animal( ) { }
Bird::Bird(string loc):Animal(loc) {}
// Constructor of the child calls the constructor of the parent
```

- The constructors of a base class are not automatically inherited in the derived class.
- One constructor of the base class must be invoked to initialize the variables in the base class
- If you don't name it, the constructor with no argument will be automatically invoked.