CSCE 121

Inheritance

Dr. Tim McGuire

Some of the material and images from Bradley Kjell, Central Connecticut State University

What Is Inheritance?

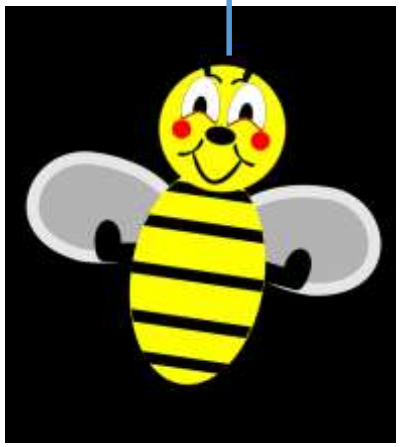
- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class

Example: Insects

Insect

All insects have certain characteristics

Images: public domain clipart



In addition to the common insect characteristics, the bee has its own unique characteristics such as the ability to sting.

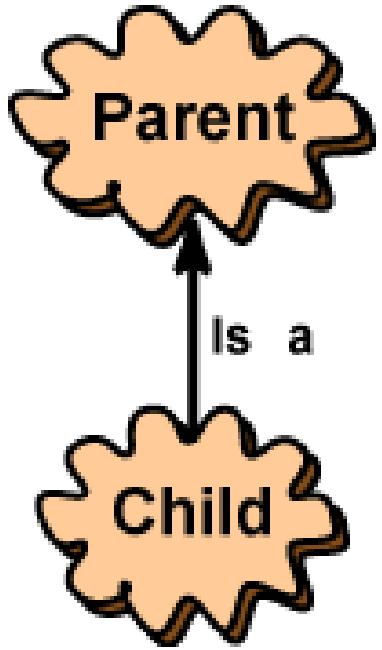
In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

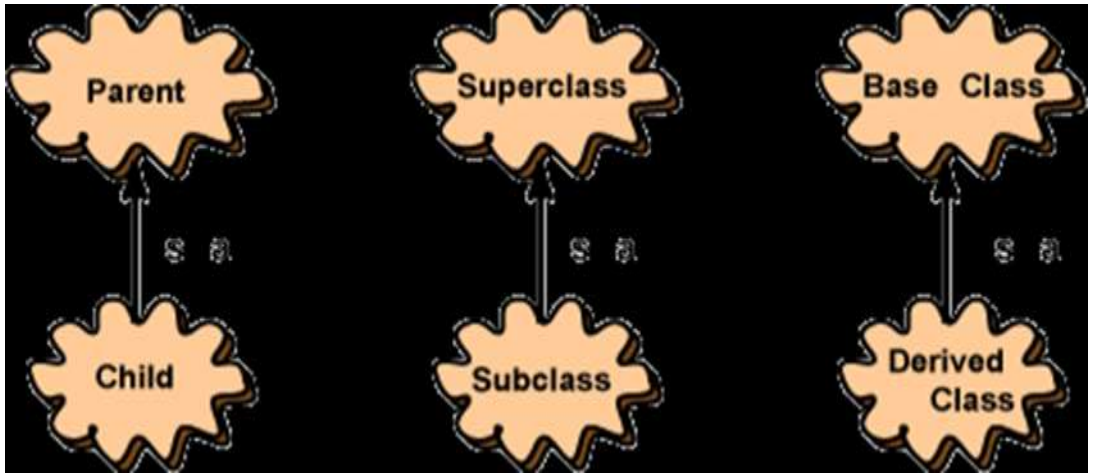
The "is a" Relationship

- Inheritance establishes an "is a" relationship between classes.
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete

Inheritance – Terminology and Notation

- The class that is used to define a new class is called a parent class (or superclass or **base** class.) The class based on the parent class is called a child class (or subclass or **derived** class.)
- In diagrams, the arrow points from the child to the parent.





Inheritance – Terminology and Notation

- Base class (or parent) – inherited from
- Derived class (or child) – inherits from the base class
- Notation:

```
class Student          // base class
{
  . . .
};
class UnderGrad : public Student
{                              // derived class
  . . .
};
```

Back to the ‘is a’ Relationship

- An object of a derived class 'is a(n)' object of the base class
- Example:
  - an `UnderGrad` is a `Student`
  - a `Mammal` is an `Animal`
- A derived object has all of the characteristics of the base class

Two Simple Questions (1)

- Can a parent class have more than one child class?
- Can a parent class inherit characteristics from its child class?

Two Simple Questions (2)

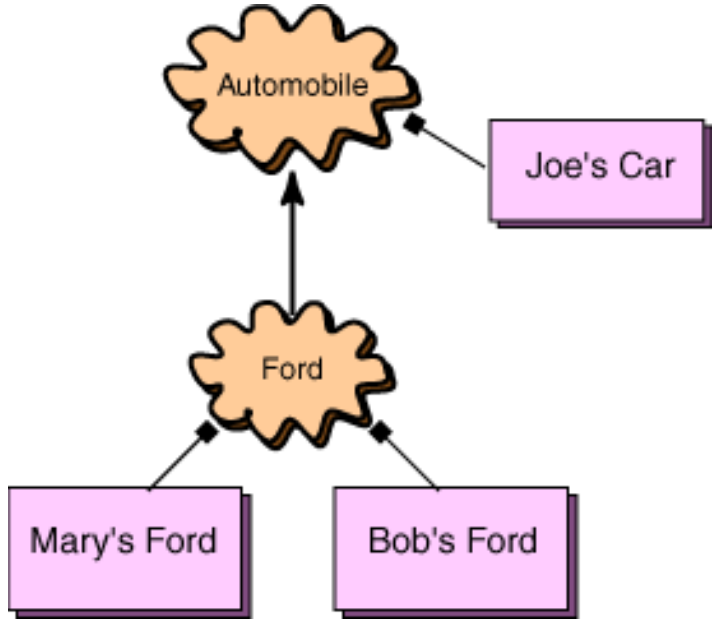
- Can a parent class have more than one child class?
  - Yes, a parent can have any number of children.
  - In C++ (but not most other object-oriented languages) a child can have more than one parent.
  - We won't address multiple inheritance in this course
- Can a parent class inherit characteristics from its child class?
  - No. Inheritance goes in only one direction.

Superclasses and Subclasses

- A superclass can have multiple subclasses.
- Subclasses can be superclasses of other subclasses.
- A big advantage of inheritance is that we can write code that is common to multiple classes once and reuse it in subclasses.
  - A subclass can define new instance variables and methods, some of which may override (hide) those of a superclass.

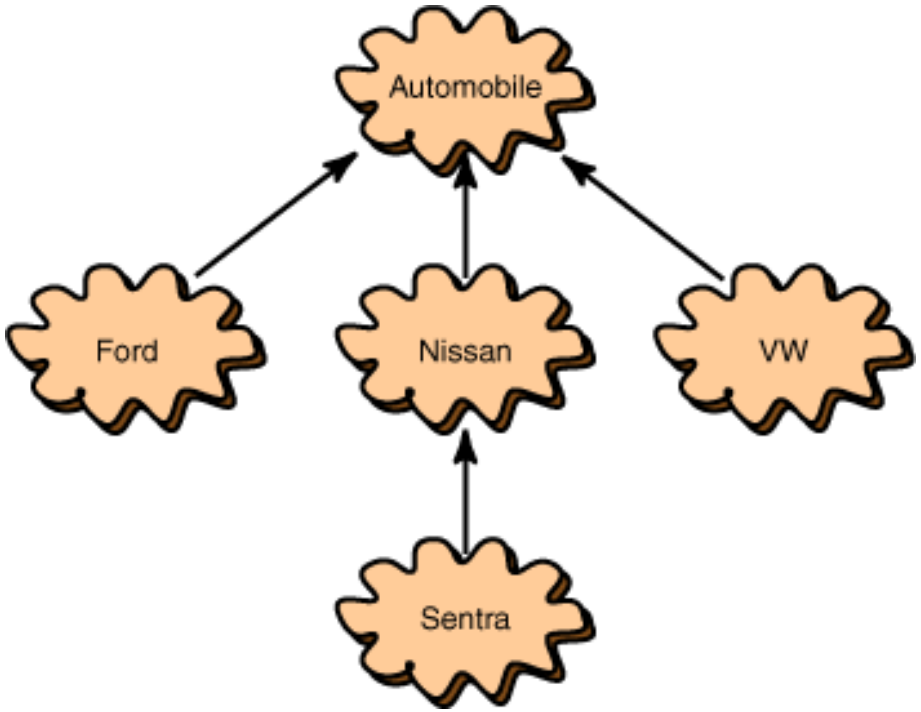
Hierarchies

- We can organize classes into hierarchies of functionality.
- The class at the top of the hierarchy (superclass) defines instance variables and methods common to all classes in the hierarchy.
- We derive a subclass, which inherits behavior and fields from the superclass.



Hierarchies (2)

- This picture shows a hierarchy of classes. It shows that:
  - "Ford is-a automobile,"
  - "Nissan is-a automobile,"
  - "VW is-a automobile."
  - It also shows that "Sentra is-a Nissan."



What Does a Child Have?

An object of the derived class has:

- all members defined in child class
- all members declared in parent class

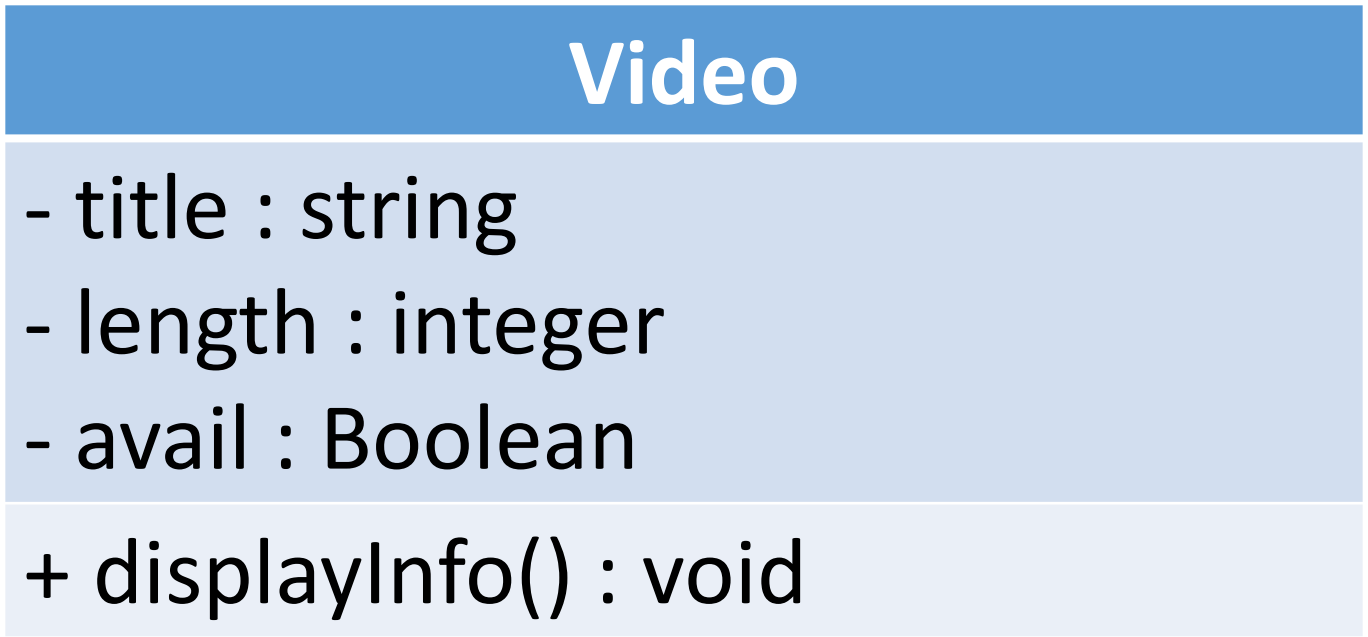
An object of the derived class can use:

- all `public` members defined in child class
- all `public` members defined in parent class

Video Example

class Video

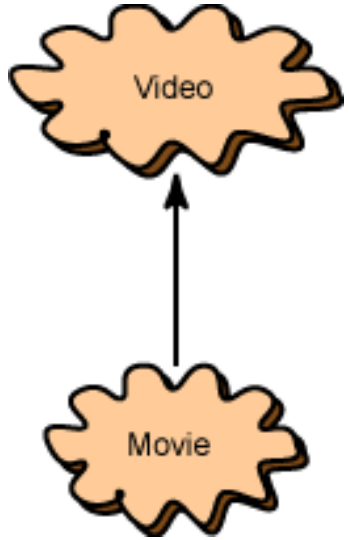
- Suppose we have a class Video to represent videos available from a streaming service. The UML diagram might look like this:



Using Inheritance

- The Video class has basic information in it, and could be used for documentaries and instructional videos. But more information is needed for movie videos. Let us make a class that is similar to Video, but now includes the name of the director and a rating.

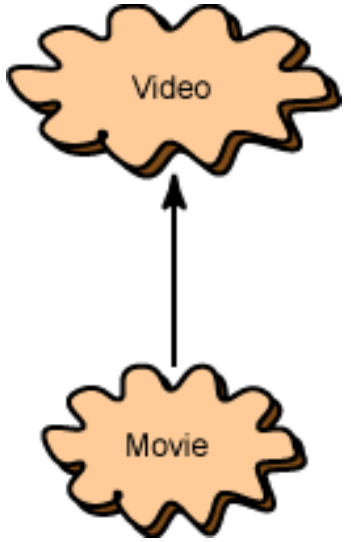
Movie	
- title : string - length : integer - avail : Boolean - director: string - rating : string	Inherited from Video Inherited from Video Inherited from Video Defined in Movie Defined in Movie
+ displayInfo() : void	Inherited from Video



Using Inheritance

- The class Movie is a subclass of Video. An object of type Movie has these members:

Movie	
- title : string	Inherited from Video
- length : integer	Inherited from Video
- avail : Boolean	Inherited from Video
- director: string	Defined in Movie
- rating : string	Defined in Movie
+ displayInfo() : void	Inherited from Video



- Both classes are defined: the Video class can be used to construct objects of that type, and now the Movie class can be used to construct objects of the Movie type.

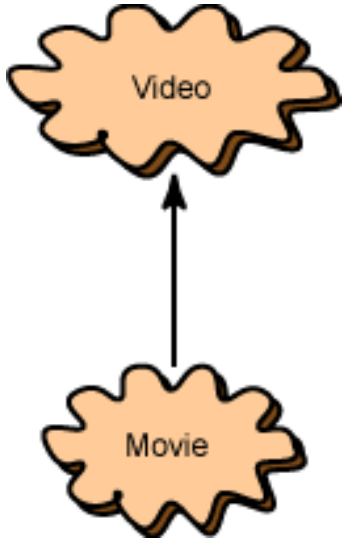
Instantiating these classes

- We can make instances of these classes now.
- A video might have:
  - Title: “Pandemic: How to Prevent an Outbreak”
  - Length: 50
- A movie might have:
  - Title: “The Princess Bride”
  - Length: 98
  - Director: “Rob Reiner”
  - Rating: “PG”

Using Inheritance

- Video does not define the instance variables director and rating, so its displayInfo() method can't use them to display those.

Movie	
- title : string	Inherited from Video
- length : integer	Inherited from Video
- avail : Boolean	Inherited from Video
- director: string	Defined in Movie
- rating : string	Defined in Movie
+ displayInfo() : void	~~Inherited from Video~~ Defined in Movie



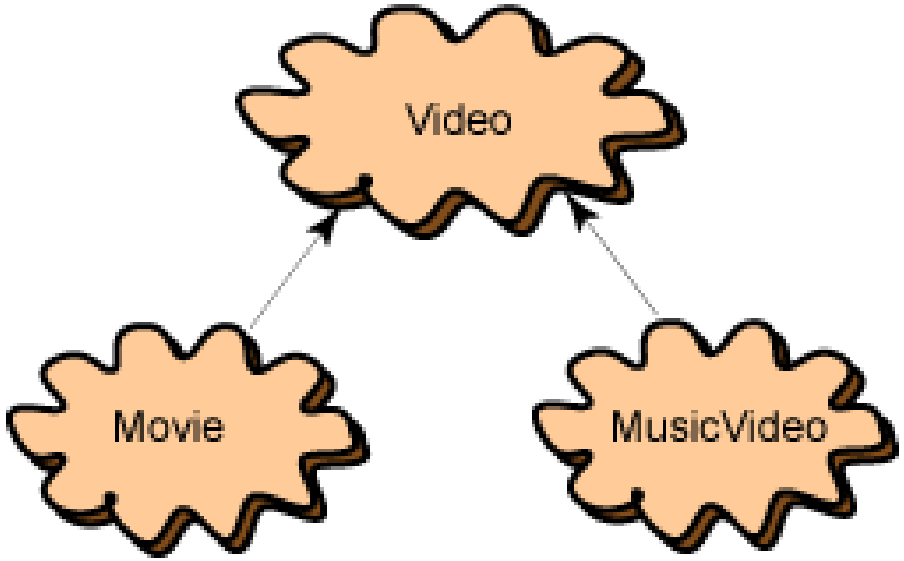
- A child's method overrides a parent's method when it has the same signature as a parent method. Now the parent has its method, and the child has its own method with the same signature.

Redefining Base Class Functions

- Not the same as overloading – with overloading, parameter lists must be different
- Objects of base class use base class version of function; objects of derived class use derived class version of function

Another Derived Class

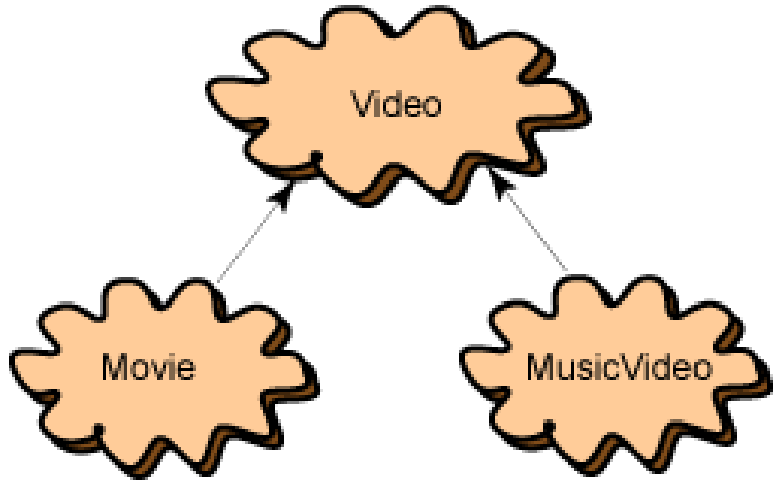
- So far the video streaming application has two classes: Video and Movie.
- Say that you wanted to create a new class, MusicVideo that will be like Video but will have two new instance variables: artist and genre ("R&B", "Pop", "Country", "Other").
- Both of these will be Strings. The MusicVideo class will need its own displayInfo() method.



Overriding Methods

- The MusicVideo class is a subclass of Video.
- It adds the member variables artist and genre

MusicVideo	
- title : string	Inherited from Video
- length : integer	Inherited from Video
- avail : Boolean	Inherited from Video
- artist: string	Defined in MusicVideo
- genre : string	Defined in MusicVideo
+ displayInfo() : void	~~Inherited from Video~~ Defined in MusicVideo



- As before, we also will need a new displayInfo() method for the MusicVideo class

C++ Example

- GradedActivity.h
- GradedActivity.cpp
- driver1.cpp

C++ Example

- GradedActivity.h
- GradedActivity.cpp
- driver1.cpp

C++ Example

- GradedActivity.h
- GradedActivity.cpp
- FinalExam.h
- FinalExam.cpp
- Driver2.cpp

C++ Example

- GradedActivity.h
- GradedActivity.cpp
- FinalExam.h
- FinalExam.cpp
- Driver2.cpp

Protected Members and Class Access

Protected Members and Class Access

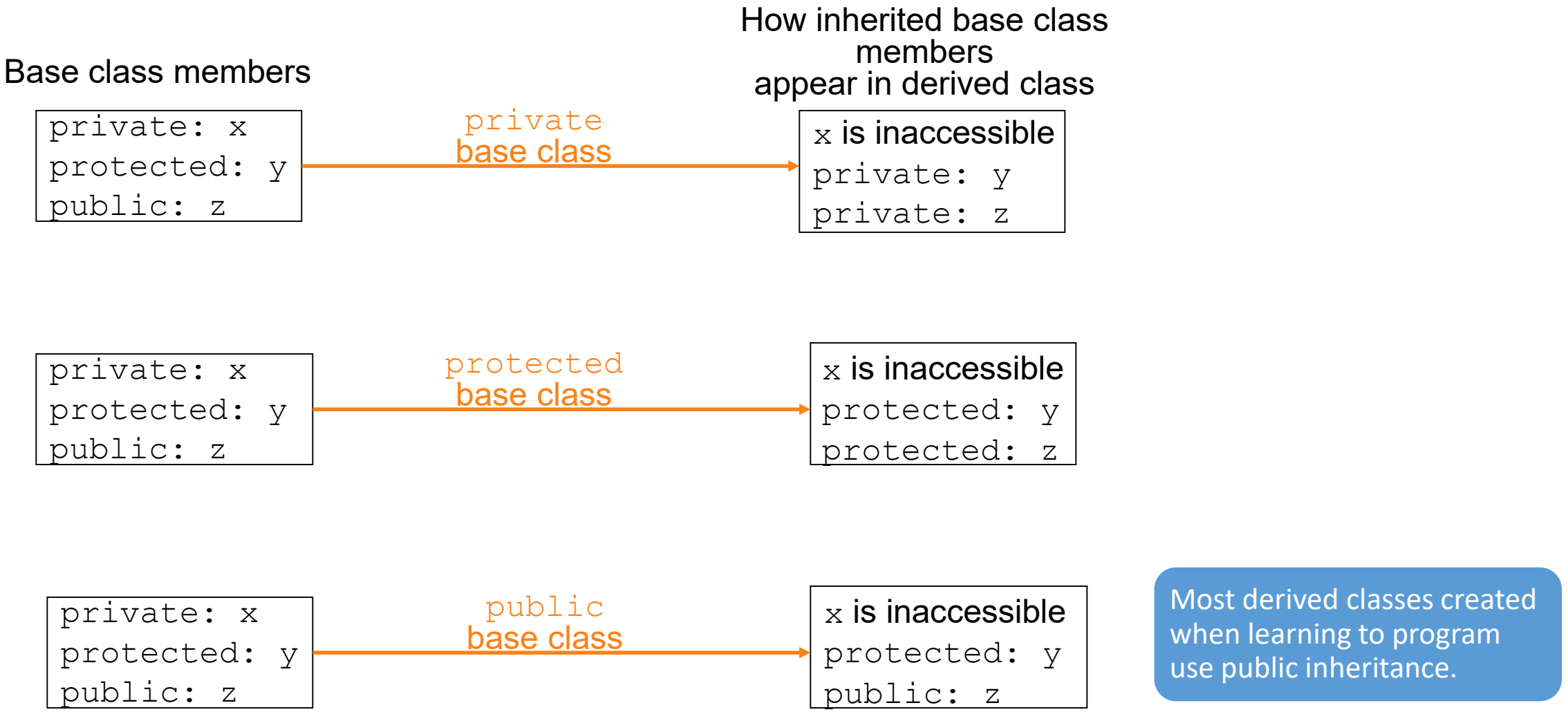
- `protected` member access specification: like `private`, but accessible by objects of derived class

- Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class

Class Access Specifiers

1) `public` – object of derived class can be treated as object of base class (not vice-versa)
2) `protected` – more restrictive than `public`, but allows derived classes to know details of parents
3) `private` – prevents objects of derived class from being treated as objects of base class.

Inheritance vs. Access



Most derived classes created when learning to program use public inheritance.

More Inheritance vs. Access

class Grade

private members:
```
  char letter;
  float score;
  void calcGrade();
```
public members:
```
  void setScore(float);
  float getScore();
  char getLetter();
```

class Test : public Grade

private members:
```
  int numQuestions;
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
```

When `Test` class inherits from `Grade` class using `public` class access, it looks like this: →

private members:
```
  int numQuestions:
  float pointsEach;
  int numMissed;
```
public members:
```
  Test(int, int);
  void setScore(float);
  float getScore();
  char getLetter();
```

Constructors and Destructors in Base and Derived Classes

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Constructor & Destructor example

- See constructors-destructors.cpp

Constructors and Destructors in Base and Derived Classes

```
1  // This program demonstrates the order in which base and
2  // derived class constructors and destructors are called.
3  #include <iostream>
4  using namespace std;
5
6  //*********************************
7  // BaseClass declaration          *
8  //*********************************
9
10 class BaseClass
11 {
12 public:
13    BaseClass()  // Constructor
14       { cout << "This is the BaseClass constructor.\n"; }
15
16    ~BaseClass() // Destructor
17       { cout << "This is the BaseClass destructor.\n"; }
18 };
```

```
20  //*********************************
21  // DerivedClass declaration       *
22  //*********************************
23
24  class DerivedClass : public BaseClass
25  {
26  public:
27     DerivedClass()  // Constructor
28        { cout << "This is the DerivedClass constructor.\n"; }
29
30     ~DerivedClass()  // Destructor
31        { cout << "This is the DerivedClass destructor.\n"; }
32  };
33
34  //*********************************
35  // main function                  *
36  //*********************************
37
38  int main()
39  {
40     cout << "We will now define a DerivedClass object.\n";
41
42     DerivedClass object;
43
44     cout << "The program is now going to end.\n";
45     return 0;
46  }
```

Program Output

```
We will now define a DerivedClass object
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

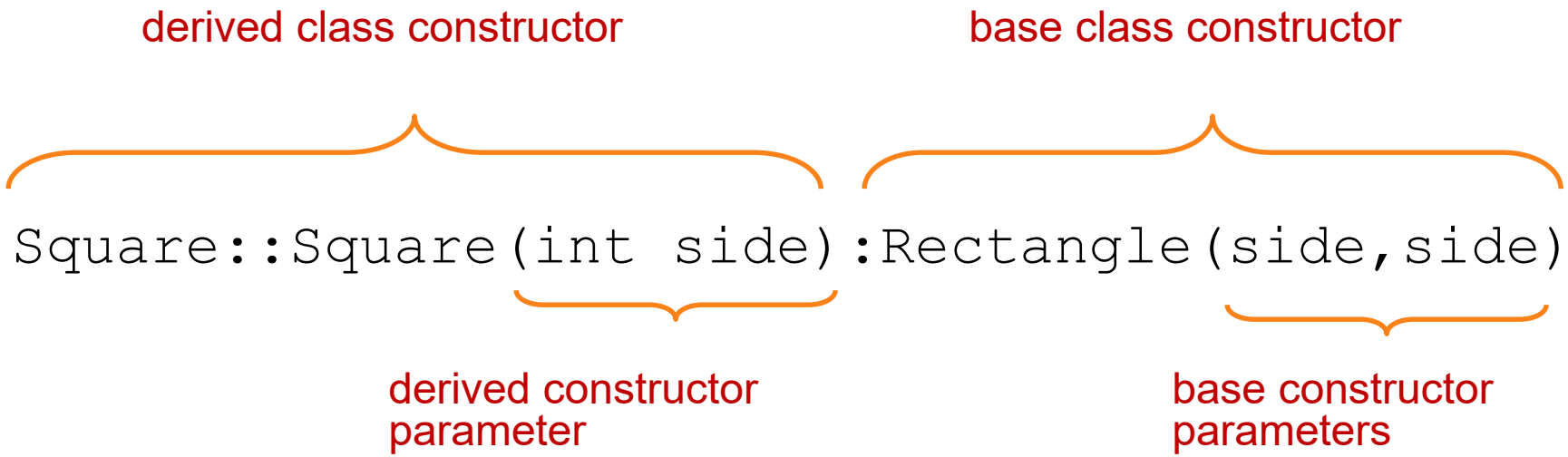
Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:

```
Square::Square(int side) : Rectangle(side, side)
```

- Must be done if base class has no default constructor

Passing Arguments to Base Class Constructor



- See
  - Rectangle.h
  - Cube.h
  - cube.cpp

Polymorphism and Virtual Member Functions

Polymorphism

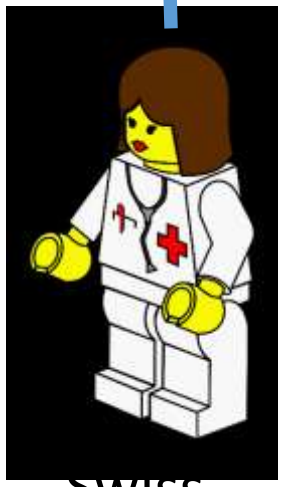
- Polymorphism refers to the ability to associate multiple meanings with one function name using a mechanism called late binding
- Polymorphism is a key component of the philosophy of object oriented programming

Subtype polymorphism

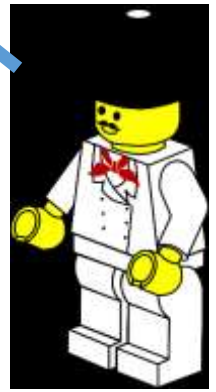
- AKA **subclassing** or **inheritance**

Person
(parent / base class)

Canadian
(child / derived class)



Swiss
(child / derived class)



French
(child / derived class)

Subtype polymorphism helps to provide simple interfaces

- We only need to know one method to use with an unknown number of sub-classes.
  - E.g. Person::hello(), defined by default as “Bonjour!”

- With polymorphism, we can use the People lens to look at each person and not have to worry about what subtype of Person they are.
  - Because every Person can do hello().

- However, this might not be very useful if we can only do what the parent class does by default.
  - Do all people do hello in the same way?

Speak!

Image source: arbabwaseer@gmail.com

Same command!

```
void NowSpeak(const Animal& animal) {
    animal.speak();
}
```

Results differ based on the type of animal.

Polymorphism is Useful

- What if we want to use the derived version instead of the base version?
  - the Canadian version of hello(), instead of the Person version
  - The Duck version of speak(), instead of the Animal version
- There needs to be a way to indicate that the derived version should ***override*** the parent version.
- When the person says “Now Speak”, each animal should speak according to what type of animal it is.
- With polymorphism, we can indicate a method (e.g. speak) and it will do what is appropriate for that type (e.g. “quack”, “meow”, “woof”).

A Late Binding Example

- Imagine a graphics program with several types of figures
  - Each figure may be an object of a different class, such as a circle, oval, rectangle, etc.
  - Each is a descendant of a class Figure
  - Each has a function draw( ) implemented with code specific to each shape
  - Class Figure has functions common to all figures

A Problem

- Suppose that class `Figure` has a function `center()`
  - Function `center()` moves a figure to the center of the screen by erasing the figure and redrawing it in the center of the screen
  - Function `center()` is inherited by each of the derived classes
    - Function `center()` uses each derived object's draw function to draw the figure
    - The `Figure` class does not know about its derived classes, so it cannot know how to draw each figure

Virtual Functions

- Because the Figure class includes a method to draw figures, but the Figure class cannot know how to draw the figures, virtual functions are used
- Making a function ***virtual*** tells the compiler that you don't know how the function is implemented and to wait until the function is used in a
program, then get the implementation from the object.
  - This is called ***late binding***

Virtual functions mark which functions can be overridden by a derived class

- Indicate in the base that the derived version should be used instead if base version is also available.
  - In C++ this is known as a virtual function

```
virtual void speak();
```

- When looking at a method invocation on an instance of a base class and the compiler sees a virtual method, it knows to look for the derived class version of the function.

- If the derived class does not have its own version, it will use the base class version.

Virtual Functions in C++

- As another example, let's design a record-keeping program for an auto parts store
  - We want a versatile program, but we do not know all the possible types of sales we might have to account for
    - Later we may add mail-order and discount sales
    - Functions to compute bills will have to be added later when we know what type of sales to add
    - To accommodate the future possibilities, we will make the bill() function a virtual function

The Sale Class

- All sales will be derived from the base class Sale
- The bill() function of the Sale class is virtual
- The member function savings() and operator < each use bill()
- The Sale class interface and implementation are shown in sale.h and sale.cpp

Sale.h

```
class Sale
{
   public:
       Sale();
       Sale(double thePrice);
       virtual double bill() const;
       double savings(const Sale& other) const;
       //Returns the savings if you buy other instead of the calling object.
   protected:
       double price;
};
bool operator < (const Sale& first, const Sale& second);
//Compares two sales to see which is larger.
```

Sale.cpp

```
#include "sale.h"

    Sale::Sale()
    { price = 0.0;}

    Sale::Sale(double thePrice)
    { price = thePrice; }

    double Sale::bill() const
    { return price; }

    double Sale::savings(const Sale& other) const
    { return ( bill() - other.bill() ); }

    bool operator < (const Sale& first, const Sale& second)
    { return (first.bill() < second.bill()); }
```

Virtual Function bill

- Because function bill() is virtual in class Sale, function savings() and operator <, defined only in the base class, can in turn use a version of bill() found in a derived class
  - When a DiscountSale object calls its savings() function, defined only in the base class, function savings() calls function bill()
  - Because bill() is a virtual function in class Sale, C++ uses the version of bill() defined in the object that called savings()

DiscountSale::bill

- Class DiscountSale has its own version of virtual function bill()
  - Even though class Sale is already compiled, Sale::savings() and Sale::operator< can still use function bill() from the DiscountSale class
  - The keyword **virtual** tells C++ to wait until bill() is used in a program to get the implementation of bill() from the calling object
  - DiscountSale is defined and used in
    - Discountsale.h
    - Discountsale.cpp

```
class DiscountSale : public Sale
   {
   public:
       DiscountSale();
       DiscountSale(double thePrice, double theDiscount);
       //Discount is expressed as a percent of the price.
       virtual double bill() const;
   protected:
       double discount;
   };
```

```
DiscountSale::DiscountSale() : Sale(), discount(0)
{}

DiscountSale::DiscountSale(double thePrice, double theDiscount)
          : Sale(thePrice), discount(theDiscount)
{}

double DiscountSale::bill() const
{
    double fraction = discount/100;
    return (1 - fraction)*price;
}
```

```
int main()
{
    Sale simple(10.00);   //One item at $10.00.
    DiscountSale discount(11.00, 10); //One item at $11.00 with a 10% discount.

    cout.setf(ios::fixed); cout.setf(ios::showpoint); cout.precision(2);

    if (discount < simple)
    {
        cout << "Discounted item is cheaper.\n";
        cout << "Savings is $" << simple.savings(discount) << endl;
    }
    else
        cout << "Discounted item is not cheaper.\n";
    return 0;
}
```

Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
- Function defined with key word `virtual`:

```
virtual void Y() {...}
```

- Supports dynamic binding: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Virtual Functions

- A virtual function is dynamically bound to calls at runtime.
- At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.

Abstract methods do not have default implementations

- Suppose there is not a meaningful version to use in the base class?

- Indicate that the method is abstract.
  - In C++ this is called a pure virtual function.

```
virtual void speak() = 0;
```

- When looking at a method invocation on an instance of a base class and the compiler sees a pure virtual method, it knows it must use the derived version.

- If the derived class does not have its own version, it will not compile.

Abstract Class

- An abstract class cannot be instantiated!
- In C++ this is accomplished by having a pure virtual function in the base class.
- This makes sense.
  - If you have a pure virtual function, there is no definition for that function, so the object cannot be created.
- Derived classes that will be instantiated MUST each define their own version of the pure virtual function(s) in the base class.

Virtual Details

- To define a function differently in a derived class and to make it virtual
  - Add keyword virtual to the function declaration in the base class
  - virtual is not needed for the function declaration in the derived class, but is often included
  - virtual is not added to the function definition
  - Virtual functions require considerable overhead so excessive use reduces program efficiency

Guidelines for subtype polymorphism

- If you want derived classes to have the **option** of overriding a function since a default is provided.
  - Make it virtual

- If you want to **force** derived classes to override a function (i.e. do not provide a default)
  - Make it pure virtual

- You cannot instantiate an abstract class.
  - Use references and pointers.

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer

Base Class Pointers

- Can define a pointer to a *base* class object
- Can assign it the address of a *derived* class object

Base Class Pointers

- Base class pointers and references only know about members of the base class
  - So, you can't use a base class pointer to call a derived class function

- Redefined functions in *derived* class will be ignored unless *base* class declares the function `virtual`

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.

- So, a virtual function is overridden, and a non-virtual function is redefined.

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.

C++17's `override` and `final` Key Words

- The `override` key word tells the compiler that the function is supposed to override a function in the base class.
- When a member function is declared with the `final` key word, it cannot be overridden in a derived class.