14.7

Aggregation

Aggregation

- Aggregation: a class is a member of a class
- Supports the modeling of 'has a' relationship between classes – enclosing class 'has a' enclosed class
- Same notation as for structures within structures

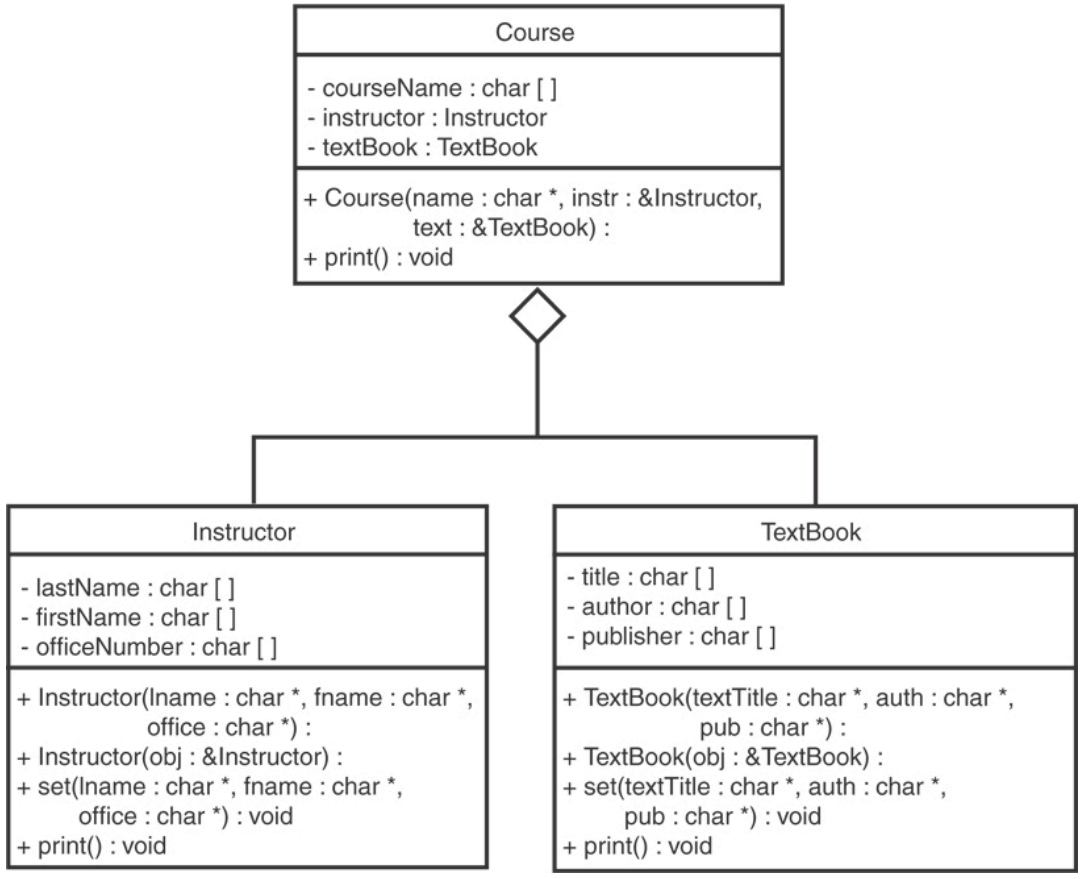
Aggregation

```
class Point
{ int x, y;

  public:
    // constructor, also default constructor
    Point(int px=0, int py =0)
    {   x = px; y = py;
    }
...
};


// Rectangle class: is comprised of two points
class Rectangle
{
  Point left_top, right_bottom;
  public:
  Rectangle()
  { left_top.setXY(0, 0);
    right_bottom.setXY(1024, 768);
  }
...
};
```

See the `Instructor`, `TextBook`, and `Course` classes in Chapter 14.



Chapter 15: Inheritance, Polymorphism, and Virtual Functions

Dr Kafi Rahman

Assistant Professor

Truman State University

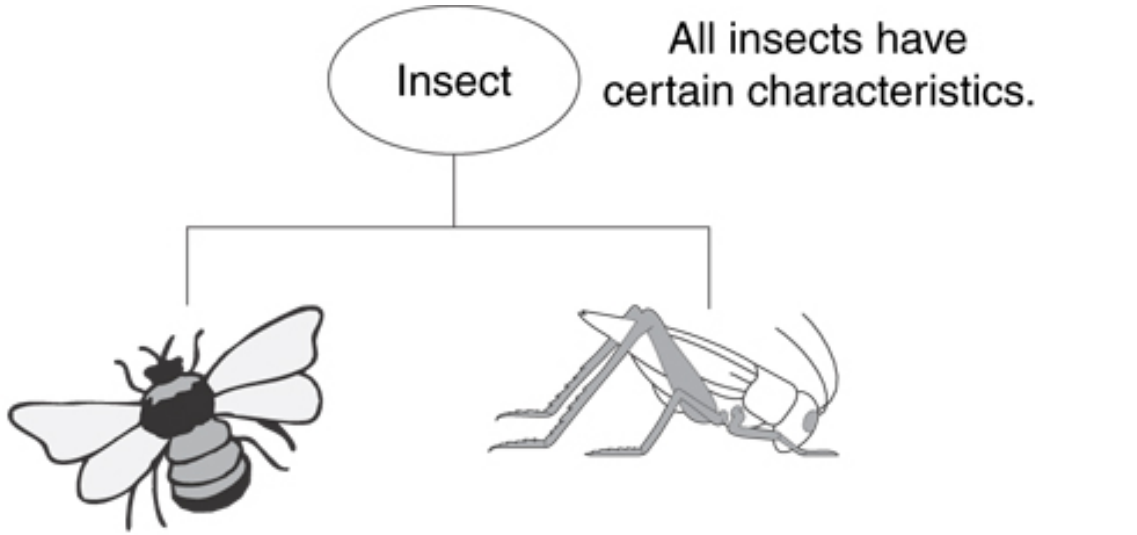
15.1

What Is Inheritance?

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



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

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

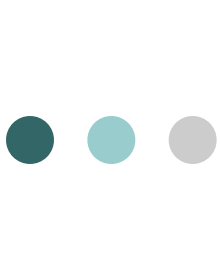
```
class Student  // base class
{
        . . .
};
class UnderGrad : public student
{       // derived class
        . . .
};
```

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

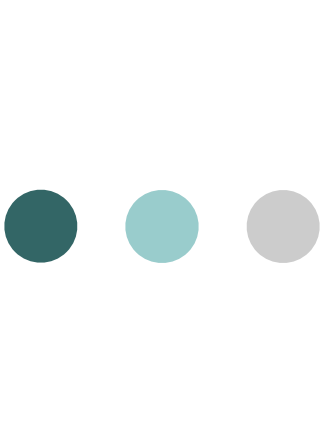
What Does a Child Have?

- An object of the derived class has:
  - all members defined in child class
  - all members declared in parent class
- Hence, an object of the derived class can use:
  - all public members defined in child class
  - all public members defined in parent class



Demo

- Let us see an example



15.2

Protected Members and Class Access

Protected Members and Class Access

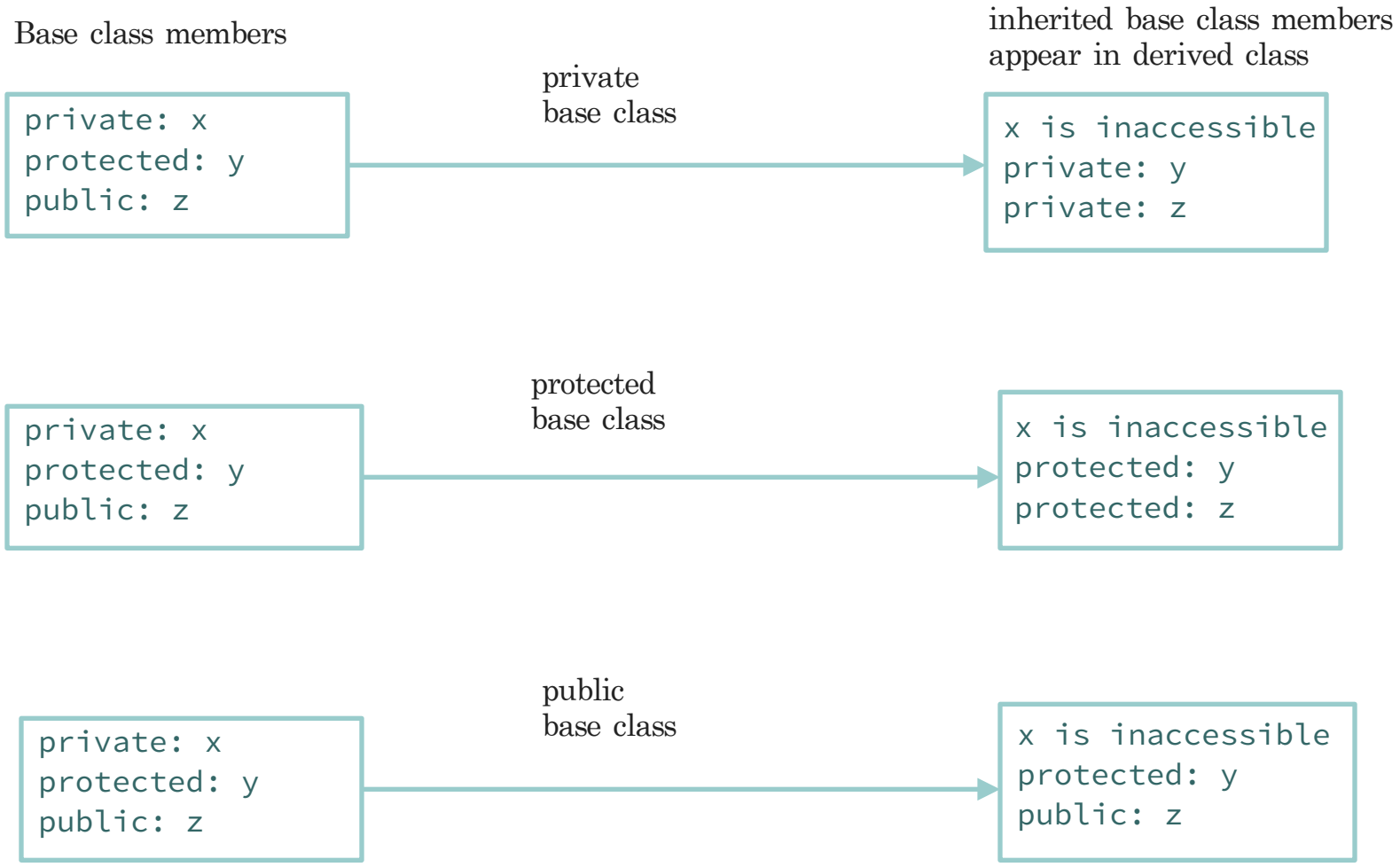
- protected member access specification: like private, but accessible by the member functions of the derived class

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

Class Access Specifiers

- public parent class resources – derived class member functions and the object of the derived class can access all of them directly (not vice-versa)
- protected parent class resources – more restrictive than public, derived class member functions can access them directly. However, not accessible to the object of the derived class.
- private parent class resources – not accessible by the member functions of the derived class or the object of the derived class.

Inheritance vs. Access



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();
  float getLetter();
```

More Inheritance vs. Access (2)

```
class Grade
private members:
  char letter;
  float score;
  void calcGrade();

public members:
  void setScore(float);
  float getScore();
  char getLetter();
```

```
class Test: protected Grade
private members:
  int numQuestions;
  float pointsEach;
  int numMissed;

public members:
  Test(int, int);
```

When Test class inherits from Grade class using protected class access, it looks like this:

```
private members:
  int numQuestions:
  float pointsEach;
  int numMissed;

public members:
  Test(int, int);
protected members:
  void setScore(float);
  float getScore();
  float getLetter();
```

More Inheritance vs. Access (3)

```
class Grade

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

```
class Test: private Grade

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

When Test class inherits from Grade class using private class access, it looks like this:

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

15.3

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

Constructors and Destructors in Base and Derived Classes

Program 15-4

```
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
```

Constructors and Destructors in Base and Derived Classes

Program 15-4 *(continued)*

```
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  };
19
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
```

Constructors and Destructors in Base and Derived Classes

```
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  }
```

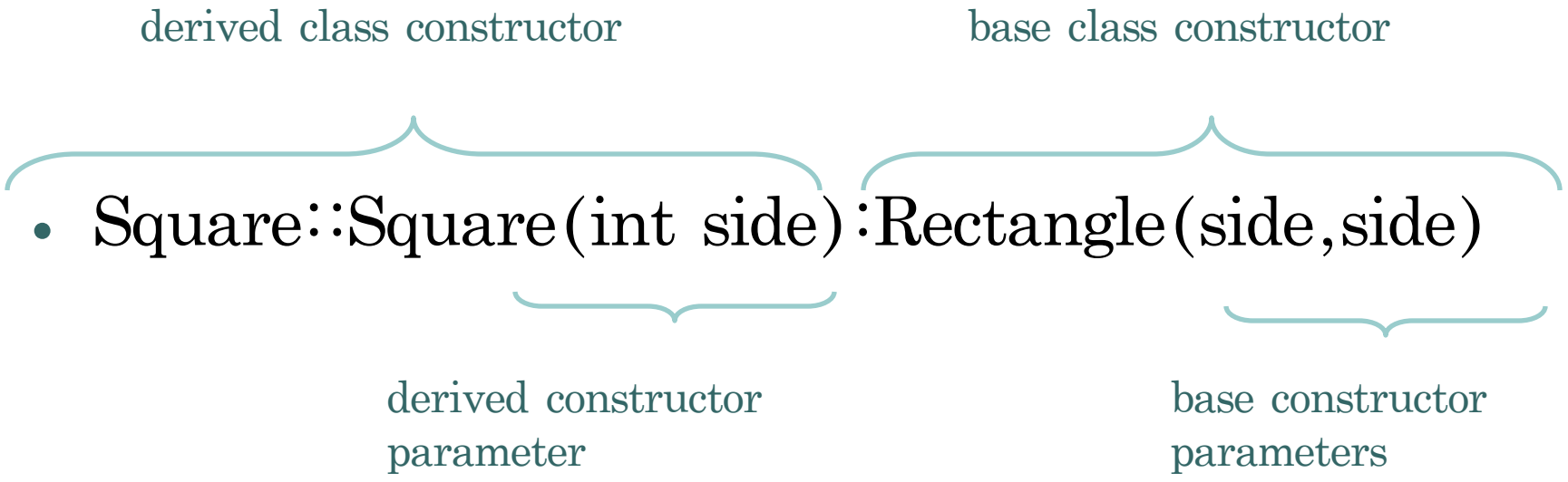
Program Output

```
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)`
- Can also be done with inline constructors
- Must be done if base class has no default constructor

Passing Arguments to Base Class Constructor



Constructor Inheritance

- In a derived class, some constructors can be inherited from the base class.
- The constructors that cannot be inherited are:
  - the default constructor
  - the copy constructor

Constructor Inheritance

- Consider the following:

```
class MyBase
{
private:
    int ival;
    double dval;
public:
    MyBase(int i)
    { ival = i; }

    MyBase(double d)
    { dval = d; }
};
```

```
class MyDerived : MyBase
{
public:
    MyDerived(int i) : MyBase(i)
    {}

    MyDerived(double d) : MyBase(d)
    {}
};
```

Constructor Inheritance

- We can rewrite the MyDerived class:

```
class MyBase
{
private:
    int ival;
    double dval;
public:
    MyBase(int i)
    { ival = i; }

    MyBase(double d)
    { dval = d; }
};
```

```
class MyDerived : MyBase
{
    using MyBase::MyBase;
};
```

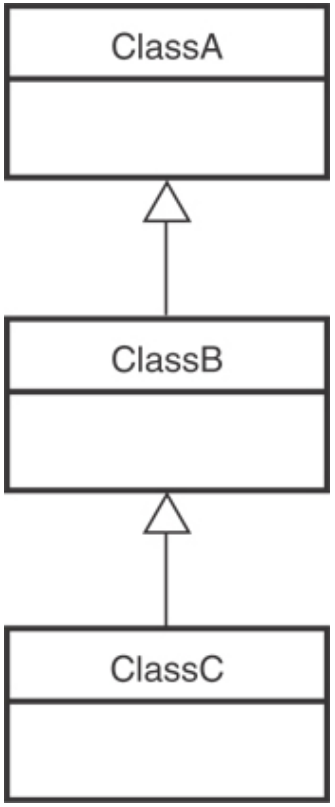
The using statement causes the class to inherit the base class constructors.

15.5

Class Hierarchies

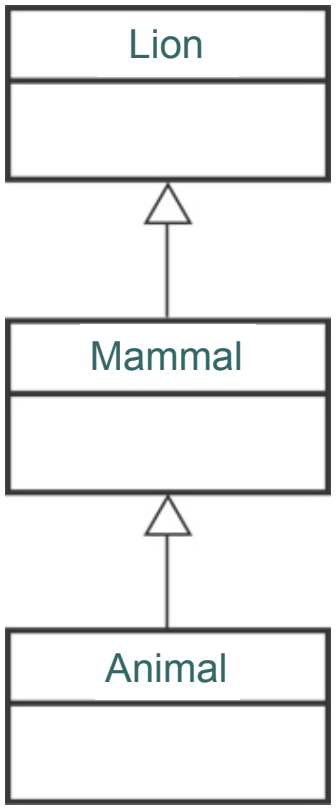
Class Hierarchies

- A base class can be derived from another base class.



Class Hierarchies (cont)

- Consider the Animal, Mammal, and the Lion class hierarchy.



Class Hierarchies (cont)

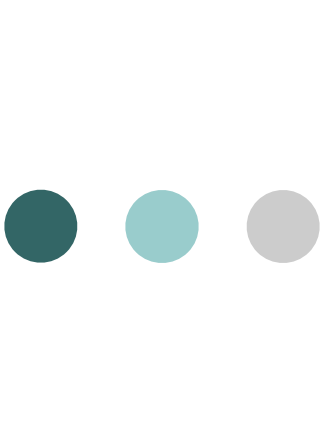
```
5   class Animal
6   {    string animalSound;
7   public:
8       Animal()
9       {    animalSound = "Awwww";
10          cout<<"\nI am an animal and I sound: "
11              <<getAnimalSound();
12      }
13      string getAnimalSound() const { return animalSound; }
14  };
```

Class Hierarchies (cont)

```
17  class Mammal : public Animal
18  {
19      string mammalSound;
20
21  public:
22      Mammal() : Animal()
23      {    mammalSound = "Meeeoowww";
24          cout<< "\nI am a Mammal and I sound: "
25              <<getMammalSound();
26      }
27      string getMammalSound() const { return mammalSound; }
28  };
```

Class Hierarchies (cont)

```
31  class Lion : public Mammal
32  {   string lionSound;
33  public:
34      Lion() : Mammal()
35      {   lionSound = "Raawwrrrrr";
36          cout<< "\nI am a Lion and I sound: "
37              <<getLionSound();
38      }
39      string getLionSound() const { return lionSound; }
40  };
41
42  int main() // testing the derived class
43  {   Lion theKing;
44      cout << "\nAnimal Sound is: " << theKing.getAnimalSound();
45      cout << "\nMammal Sound is: " << theKing.getMammalSound();
46      cout << "\nLion Sound is: " << theKing.getLionSound();
47  }
```



15.4

Redefining Base Class Functions

Redefining Base Class Functions

- Redefining function: function in a derived class that has the same name and parameter list as a function in the base class

- Typically used to replace a function in base class with different actions in derived class

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

Redefining Base Class Functions: Base Class

```
#include <iostream>
using namespace std;
// override a function

class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and I sound: "
            <<getMammalSound();
    }
    string getMammalSound() const { return mammalSound; }
};

```

Redefining Base Class Functions: Derived Class

```
19  class Lion : public Mammal
20  {
21      string lionSound;
22  public:
23      Lion() : Mammal()
24      {   lionSound = "Raawwrrrrr";
25      }
26      void display()
27      {   cout<< "\nI am a Lion and I sound: "
28              <<getLionSound();
29      }
30      string getLionSound() const { return lionSound; }
31  };
```

Redefining Base Class Functions

```
33  // using the derived class
34  int main()
35  {    Lion theKing;
36       theKing.display();
37       //theKing.Mammal::display();
38  }
39
40
41  /* Output of the program: */
42       I am a Lion and I sound: Raawwrrrrr
```

Problem with Redefining

- Consider this situation:
  - Class BaseClass defines
    - functions `x()` and `y()`
    - `x()` calls `y()`
  - Class DerivedClass inherits from BaseClass and redefines function `y()`
  - An object D of class DerivedClass is created and function `x()` is called
  - When `x()` is called then which `y()` function is used, the one defined in BaseClass or the redefined one in DerivedClass?

Problem with Redefining

BaseClass

```
void X();
void Y();
```

DerivedClass

```
void Y();
```

```
DerivedClass D;
D.X();
```

Object D invokes function X() in BaseClass

Function X() invokes function Y() in BaseClass and not function Y() in DerivedClass

Function calls are bound at compile time. This is static binding.

Problem with Redefining

```
5   class Mammal
6   {   string mammalSound;
7   public:
8       Mammal()
9       {   mammalSound = "Meeeoowww";
10      }
11      void display()
12      {   cout<< "\nI am a Mammal and I sound: "
13              <<getSound();
14      }
15      string getSound() const { return mammalSound; }
16  };
```

Problem with Redefining ..

```
19  class Lion : public Mammal
20  {
21      string lionSound;
22  public:
23      Lion() : Mammal()
24      {   lionSound = "Raawwrrrrr";
25      }
26      string getSound() const { return lionSound; }
27  };
28
29  // using the derived class
30  int main()
31  {   Lion theKing;
32      theKing.display();
33  }
```

15.6

Polymorphism and Virtual Member Functions

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 object to the appropriate version of the function.

Virtual Functions (cont)

- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

  virtual string getSound() const

- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.

- The function also becomes virtual in all derived classes automatically!
  - We do not need to redefine it to be a virtual function in the derived version of the function

Virtual Functions: Base Class

```
5   class Mammal
6   {    string mammalSound;
7   public:
8       Mammal()
9       {    mammalSound = "Meeeoowww";
10      }
11      void display()
12      {    cout<< "\nI am a Mammal and I sound: "
13               <<getSound();
14      }
15      virtual string getSound() const { return mammalSound; }
16  };
```

Virtual Functions: Derived Class

```
19  class Lion : public Mammal
20  {
21      string lionSound;
22  public:
23      Lion() : Mammal()
24      {   lionSound = "Raawwrrrrr";
25      }
26      string getSound() const { return lionSound; }
27  };
28
29  // using the derived class
30  int main()
31  {   Lion theKing;
32      theKing.display();
33  }
```

Polymorphism: Base Class Object

- In object oriented programming, a derived class object can be assigned to the object of the base class

  - `DerivedClass xDerived;`
  - `BaseClass xBase = xDerived; // legal`

Polymorphism: Base Class Object

```
class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and "
            <<"I sound: "<<getSound();
    }
    virtual string getSound() const
    { return mammalSound;
    }
};
```

```
class Lion : public Mammal
{   string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    string getSound() const
    { return lionSound;
    }
    void display()
    {   cout<< "\nI am a Lion and "
            <<"I sound: "<<getSound();
    }
};
```

Polymorphism: Base Class Object

```
51  // using the parent object
52  int main()
53  {    Lion theKing;
54       theKing.display();
55
56       // assiging the child to the parent object
57       Mammal * varMammal = &theKing;
58       varMammal->display();
59  }
60
61  /* Output of the program:
62       I am a Lion and I sound: Raawwrrrrr
63       I am a Mammal and I sound: Raawwrrrrr
64  */
```

Polymorphism: Base Class Object

- We can define a pointer to a base class object
  - and assign it the address of a derived class object
- Base class object knows about members of the base class
  - So, we can't use a base class object to call a function that is defined in the derived class
- Redefined functions in derived class will be ignored unless base class declares the function virtual

Polymorphism: Multiple child class objects

```
class Lion : public Mammal
{   string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    string getSound() const
    { return lionSound;
    }
    void display()
    {   cout<< "\nI am a Lion and "
            <<"I sound: "<<getSound();
    }
};
```

```
class Dog : public Mammal
{   string dogSound;
public:
    Dog() : Mammal()
    {   dogSound = "Baarrkkkk";
    }
    string getSound() const
    { return dogSound;
    }
    void display()
    {   cout<< "\nI am a Dog and "
            <<"I sound: "<<getSound();
    }
};
```

Polymorphism: Multiple child class objects

```
// using the parent object
int main()
{   Lion theKing;
    theKing.display();

    Dog thePet;
    thePet.display();

    // assigning child to the parent object
    Mammal * varMammal = &theKing;
    varMammal->display();
    // assigning another child
    varMammal = &thePet;
    varMammal->display();
}
```

```
/* Output of the program: */
    I am a Lion and I sound: Raawwrrrrr
    I am a Dog and I sound: Baarrkkkk
    I am a Mammal and I sound: Raawwrrrrr
    I am a Mammal and I sound: Baarrkkkk
```

Polymorphism: Definition

- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Polymorphism: Requires References or Pointers

- Polymorphic behavior is only possible when a derived class object is used as a reference variable or a pointer.

Polymorphism:
Consider again this example

```
class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and "
            <<"I sound: "<<getSound();
    }
    virtual string getSound() const
    { return mammalSound;
    }
};
```

```
class Lion : public Mammal
{   string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    string getSound() const
    { return lionSound;
    }
    void display()
    {   cout<< "\nI am a Lion and "
            <<"I sound: "<<getSound();
    }
};
```

Polymorphism: Requires References or Pointers

```
// using the parent object
int main()
{   Lion theKing;
    // assiging the derived to the usual parent object
    Mammal varMammal = theKing;
    varMammal.display();

    // assiging the derived to the parent object as reference
    Mammal &refMammal = theKing;
    refMammal.display();

    // assigning the derived to the parent as address
    Mammal * pointMammal = &theKing;
    pointMammal->display();
}
```

```
/* Output of the program */
I am a Mammal and I sound: Meeeoowww
I am a Mammal and I sound: Raawwrrrrr
I am a Mammal and I sound: Raawwrrrrr
```

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.

C++ 11'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.
- overrideKeyword.cpp and finalKeyword.cpp programs

15.7

Abstract Base Classes and Pure Virtual Functions

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:
  - `virtual void Y() = 0;`
- The = 0 indicates a pure virtual function
- Must have no function definition in the base class

Abstract Base Classes and Pure Virtual Functions

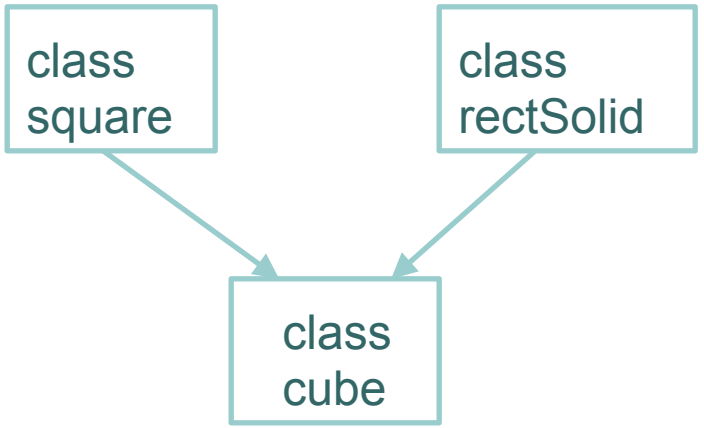
- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

15.8

Multiple Inheritance

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:
  - class cube : public square, public rectSolid;



Multiple Inheritance

- Arguments can be passed to both base classes' constructors:
  - `cube::cube(int side) : square(side), rectSolid(side, side, side);`
- Base class constructors are called in order given in class declaration, not in order used in class constructor

Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
  - Derived class redefines the multiply-defined function
  - Derived class invokes member function in a particular base class using scope resolution operator ::
- Compiler errors occur if derived class uses base class function without one of these solutions