Chapter 6

Inheritance

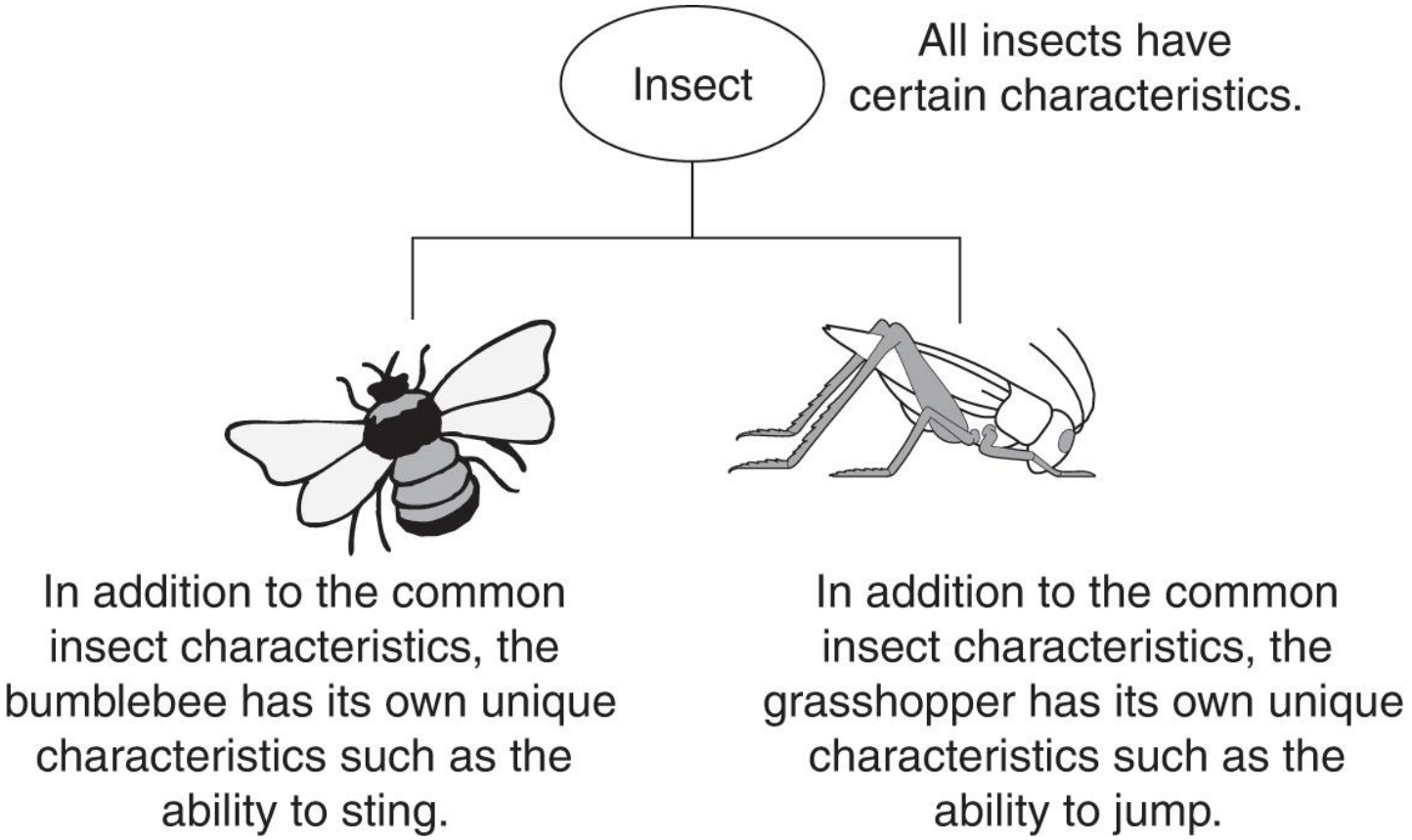
6.1

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

An object of the derived class can use:

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

15.2

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

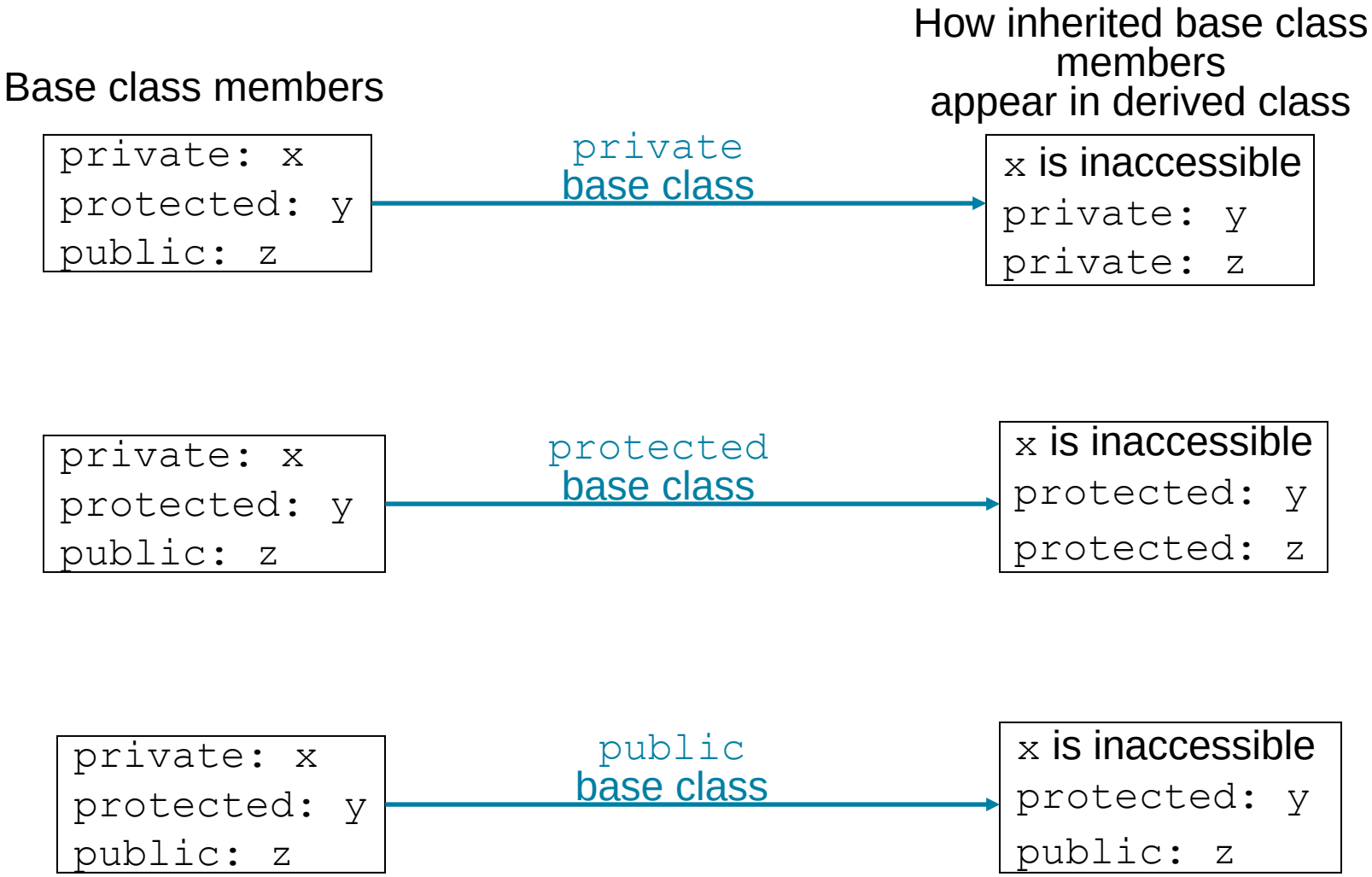
Class Access Specifiers

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



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
```

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
```

Program 15-4 (Continued)

```
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.
```

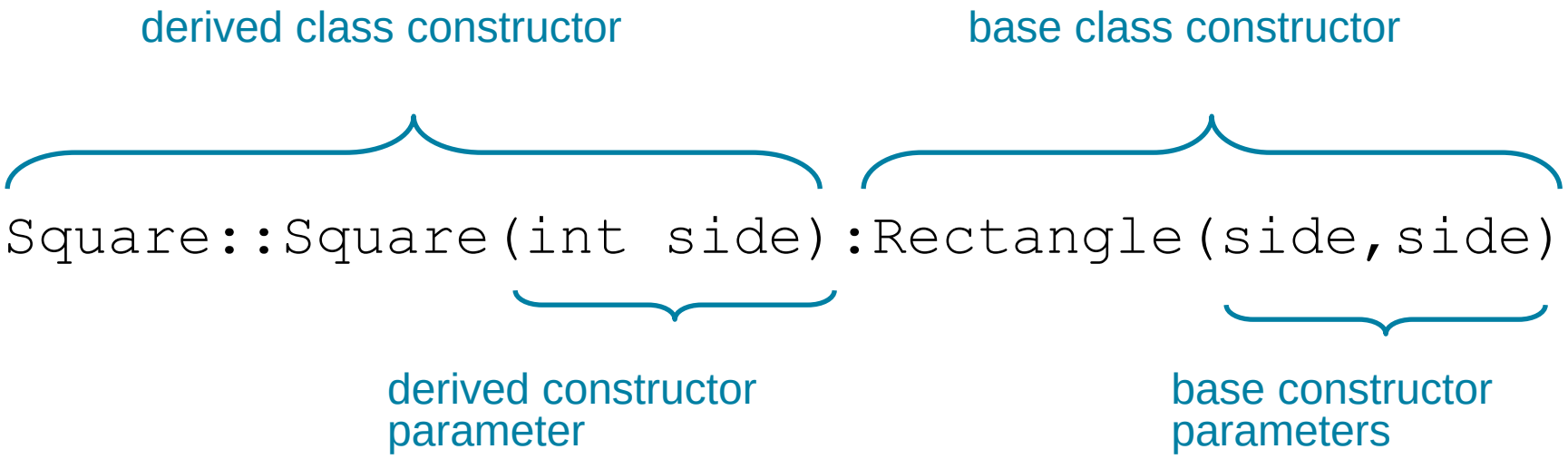
Passing Arguments to Base Class Constructor

- 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
  - the move 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 as:

```
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.

15.4

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

Base Class

```
class GradedActivity
{
protected:
   char letter;              // To hold the letter grade
   double score;             // To hold the numeric score
   void determineGrade();    // Determines the letter grade
public:
   // Default constructor
   GradedActivity()
      { letter = ' '; score = 0.0; }

   // Mutator function
   void setScore(double s)
      { score = s;
        determineGrade();}

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const
      { return letter; }
};
```

Note `setScore` function

Derived Class

```
1  #ifndef CURVEDACTIVITY_H
2  #define CURVEDACTIVITY_H
3  #include "GradedActivity.h"
4
5  class CurvedActivity : public GradedActivity
6  {
7  protected:
8     double rawScore;      // Unadjusted score
9     double percentage;    // Curve percentage
10 public:
11    // Default constructor
12    CurvedActivity() : GradedActivity()
13       { rawScore = 0.0; percentage = 0.0; }
14
15    // Mutator functions
16    void setScore(double s)
17       { rawScore = s;
18         GradedActivity::setScore(rawScore * percentage); }
19
20    void setPercentage(double c)
21       { percentage = c; }
22
23    // Accessor functions
24    double getPercentage() const
25       { return percentage; }
26
27    double getRawScore() const
28       { return rawScore; }
29 };
30 #endif
```

Redefined `setScore` function

From Program 15-7

```
13      // Define a CurvedActivity object.
14      CurvedActivity exam;
15
16      // Get the unadjusted score.
17      cout << "Enter the student's raw numeric score: ";
18      cin >> numericScore;
19
20      // Get the curve percentage.
21      cout << "Enter the curve percentage for this student: ";
22      cin >> percentage;
23
24      // Send the values to the exam object.
25      exam.setPercentage(percentage);
26      exam.setScore(numericScore);
27
28      // Display the grade data.
29      cout << fixed << setprecision(2);
30      cout << "The raw score is "
31           << exam.getRawScore() << endl;
32      cout << "The curved score is "
33           << exam.getScore() << endl;
34      cout << "The curved grade is "
35           << exam.getLetterGrade() << endl;
```

Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage for this student: 1.06 [Enter]
The raw score is 87.00
The curved score is 92.22
The curved grade is A
```

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, which `y()` is used, the one defined in `BaseClass` or the the redefined one in `DerivedClass`?

Problem with Redefining

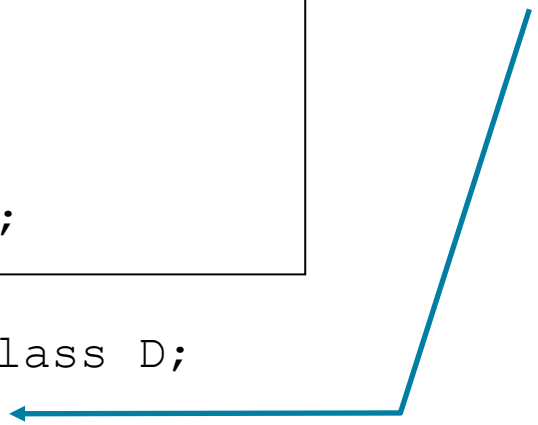
BaseClass

```
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.