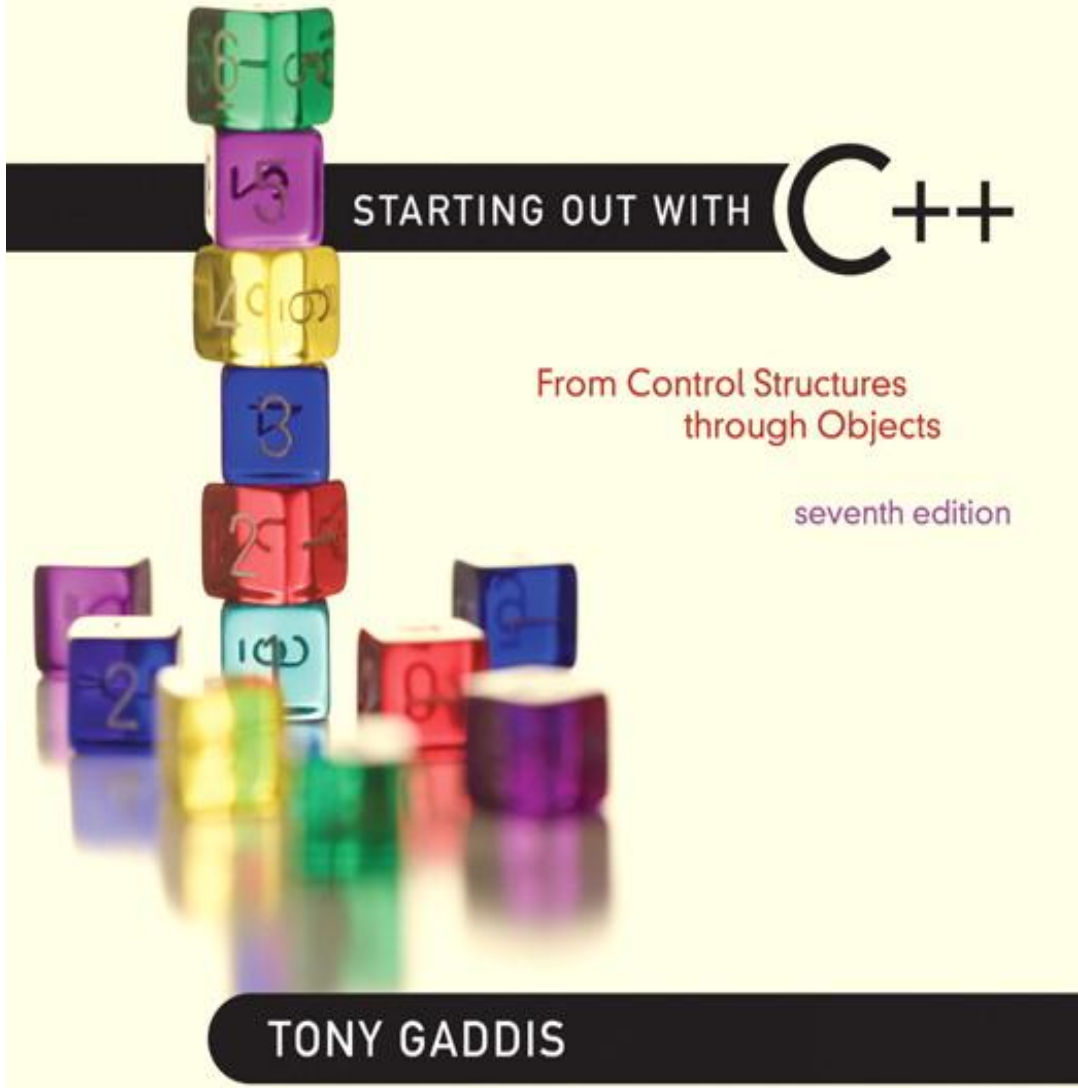
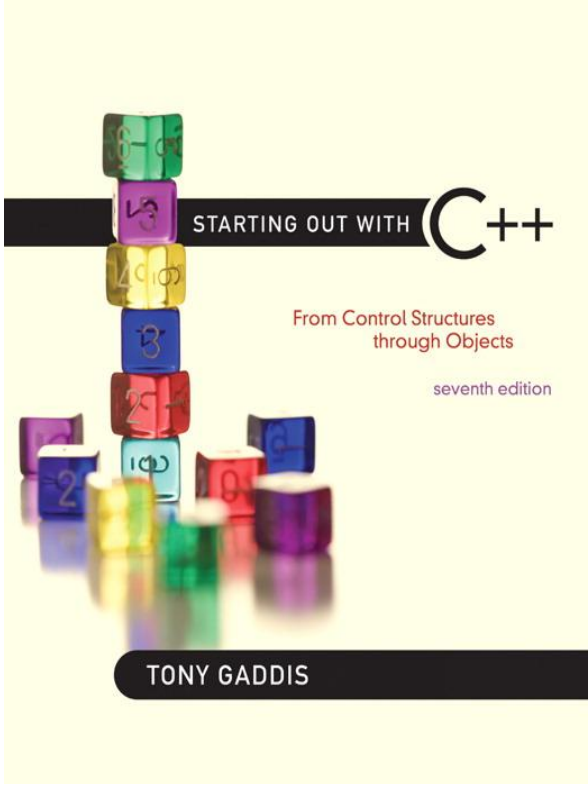
Chapter 15:

Inheritance, Polymorphism, and Virtual Functions



Addison-Wesley
is an imprint of
PEARSON



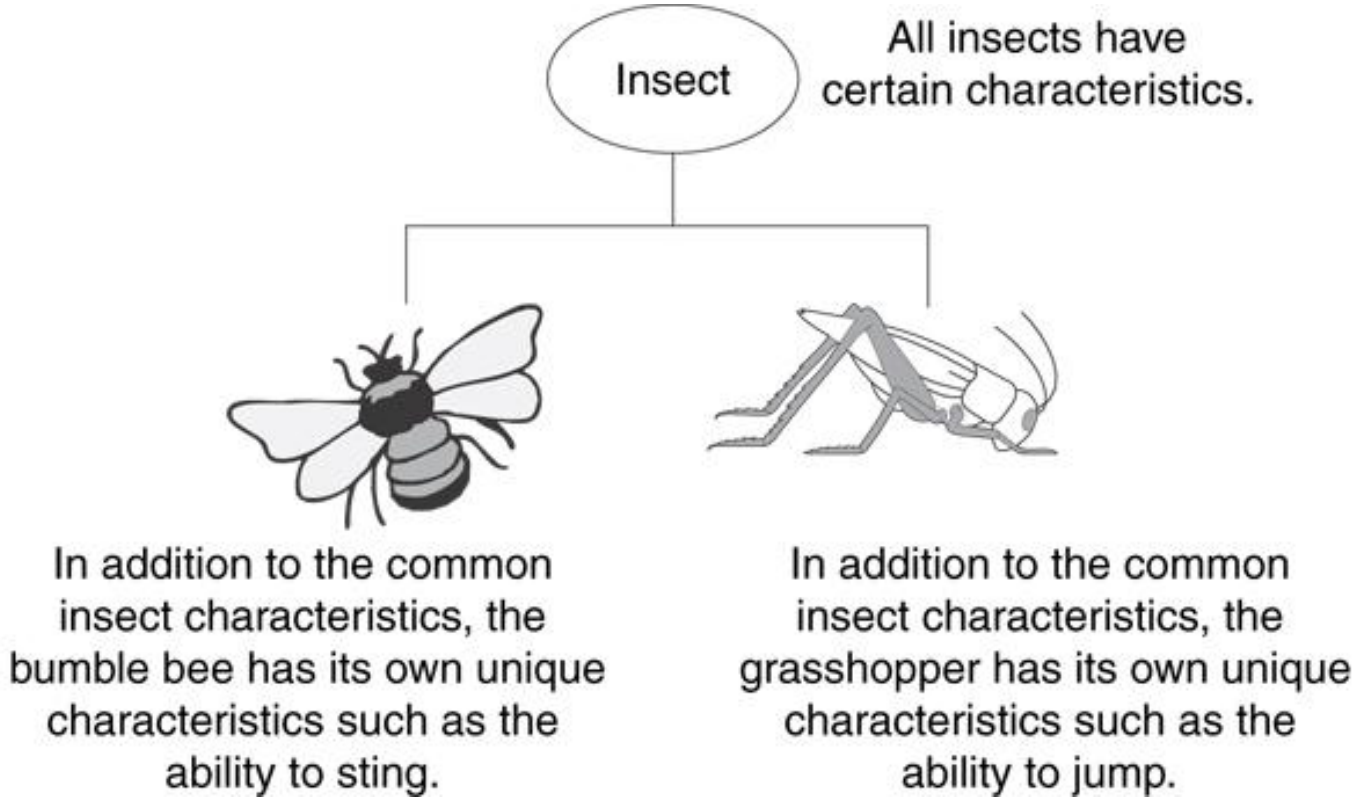
15.1

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

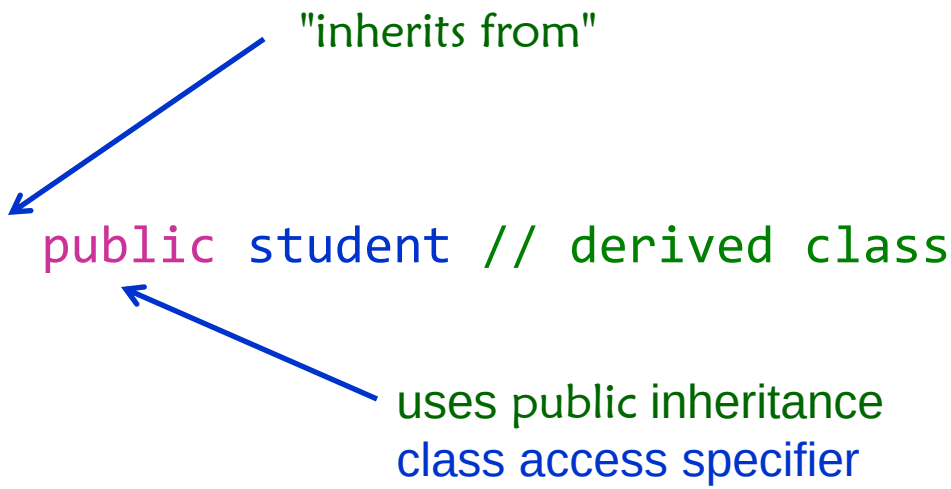
- Inheritance establishes an "is a" relationship between classes:
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is a athlete

Inheritance – Terminology and Notation

- Base class parent superclass
- Derived class child subclass
- Notation:

```
class Student // base class
{
  . . .
};

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



Graded Activity Version 1

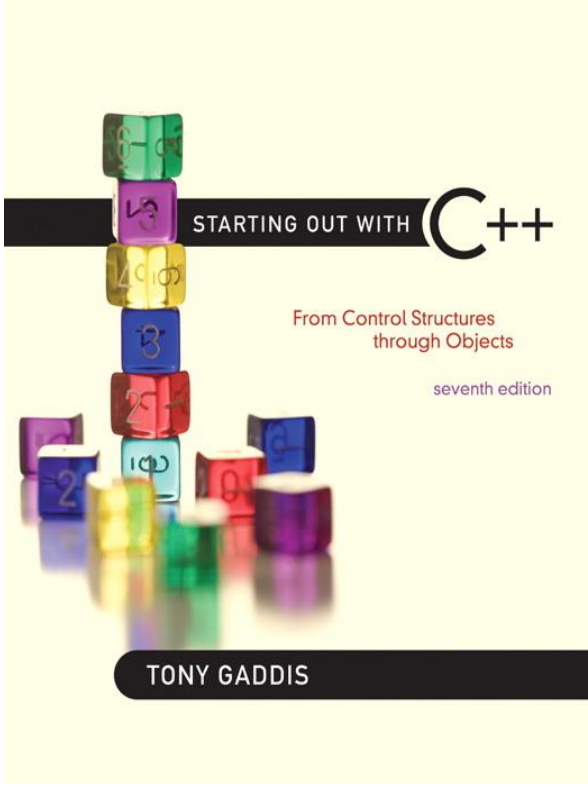
- pr 15-1
- pr15-2

Back to the 'is a' Relationship

- An object of a derived class 'is a' object of the base class
- Example:
  - an UnderGrad is a Student
  - a Mammal is a 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 the child class
  - all members declared in the parent class
- An object of the derived class can use:
  - all public members defined in the child class
  - all public members defined in the parent class
- Note: derived classes do not inherit:
  - base class constructors or destructors
  - overloaded operators



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 specifier
  - determines how private, protected, and public members of base class are inherited by the derived class

Class Access Specifiers

- public

  An object of a derived class can be treated as an object of the base class (not vice-versa)

- protected

  More restrictive than public, but it allows derived classes to know the details of the parents

- private

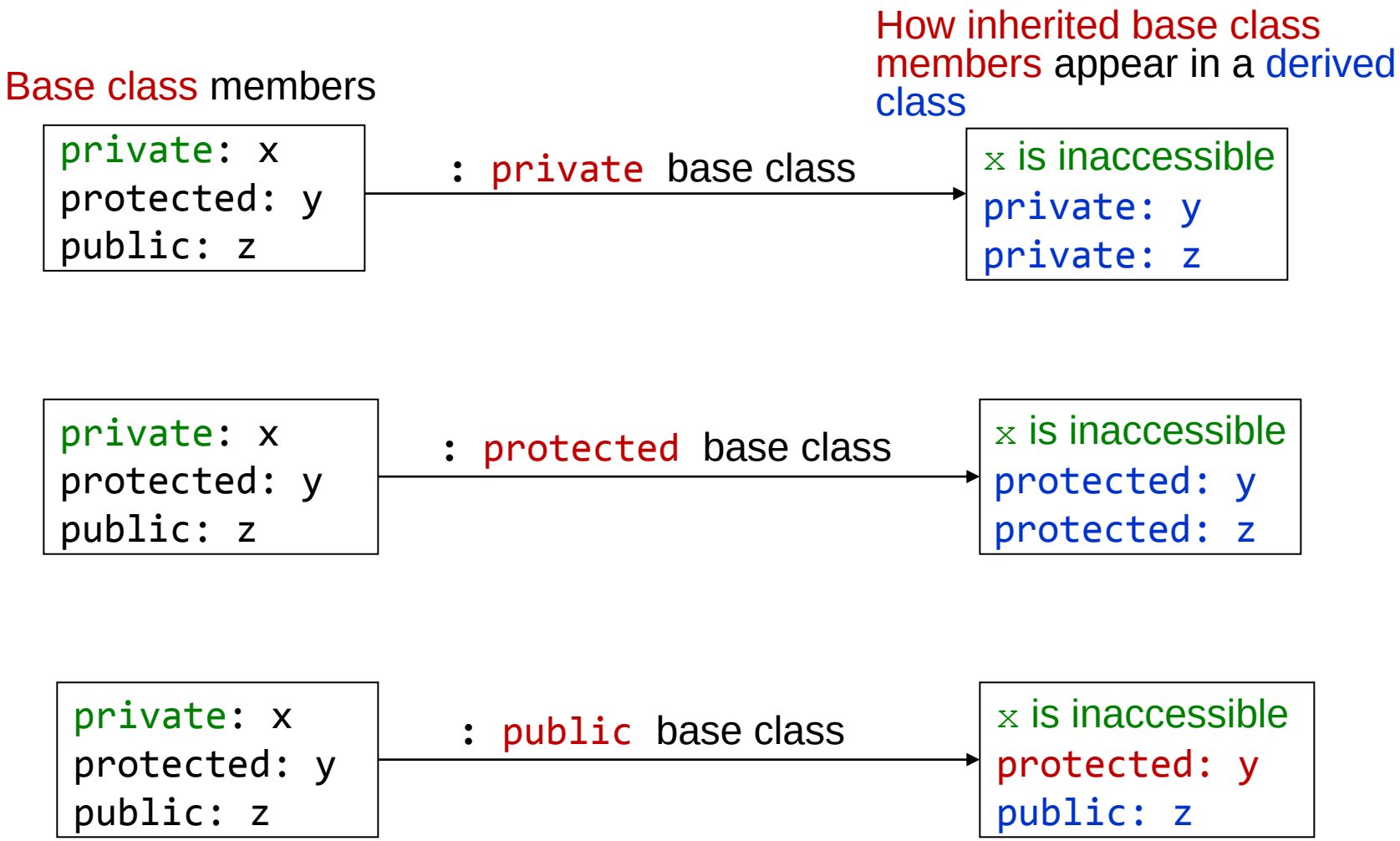
  Prevents objects of derived class from being treated as objects of base class.

Graded Activity Version 2

- pr 15-3

Inheritance vs. Access

```
class UnderGrad : public student
```



More Inheritance vs. Access

class Grade

```
private members:
  char   letter;
  double score;
  void   calcGrade();

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

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

class Test : public Grade

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

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

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

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

```
  char   letter;
  double score;
  void   calcGrade();
```

Only member data (fields) are stored in an object. One copy of the executable code for the functions (methods) is shared by all of the objects.

More Inheritance vs. Access (2)

class Grade

```
private members:
  char   letter;
  double score;
  void   calcGrade();

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

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

class Test : protected Grade

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

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

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

protected members:
  void   setScore(double);
  double getScore();
  double getLetter();

public members:
  Test(int, int);

  char   letter;
  double score;
  void   calcGrade();
```

More Inheritance vs. Access (3)

class Grade

```
private members:
  char   letter;
  double score;
  void   calcGrade();

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

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

class Test : private Grade

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

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

```
private members:
  int    numQuestions:
  double pointsEach;
  int    numMissed;
  void   setScore(double);
  double getScore();
  double getLetter();

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

```
  char   letter;
  double score;
  void   calcGrade();
```

More Inheritance vs. Access (2) (original version)

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) (original version)

class Grade

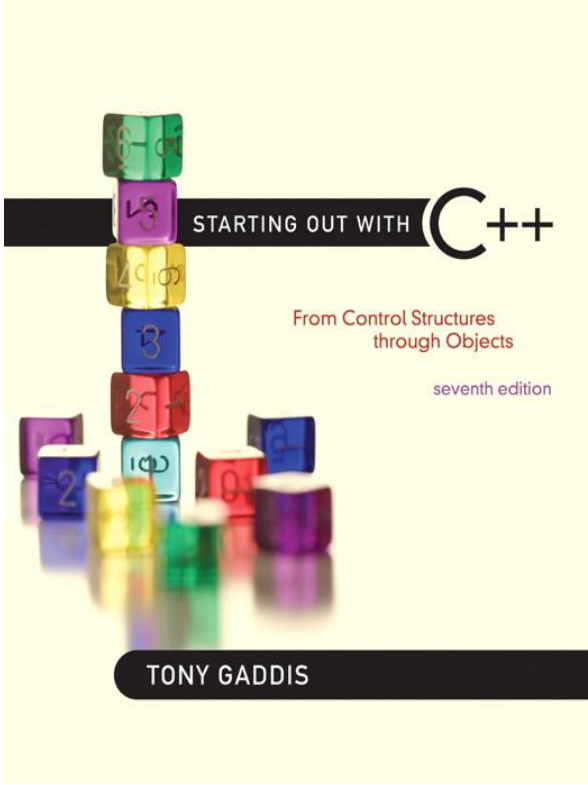
private members:
`char letter;`
`float score;`
`void calcGrade();`
public members:
`void setScore(float);`
`float getScore();`
`char getLetter();`

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 5-14 (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.
```

Base Class / Derived Class

- pr 15-4

Rectangle / Cube Classes

- pr 15-5

Automobile, Car, Truck, SUV Classes

- pr 15-6

Passing Arguments to Base Class Constructor

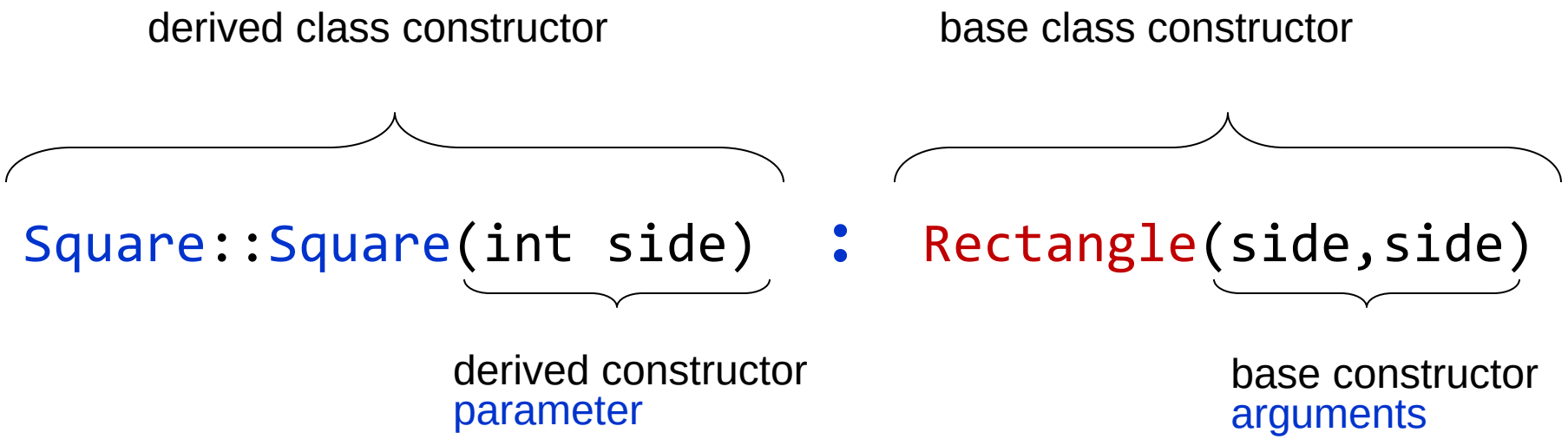
- Allows selection between multiple base class constructors (overloaded constructors)
- Specify arguments for the base constructor (superclass) in the derived constructor heading:

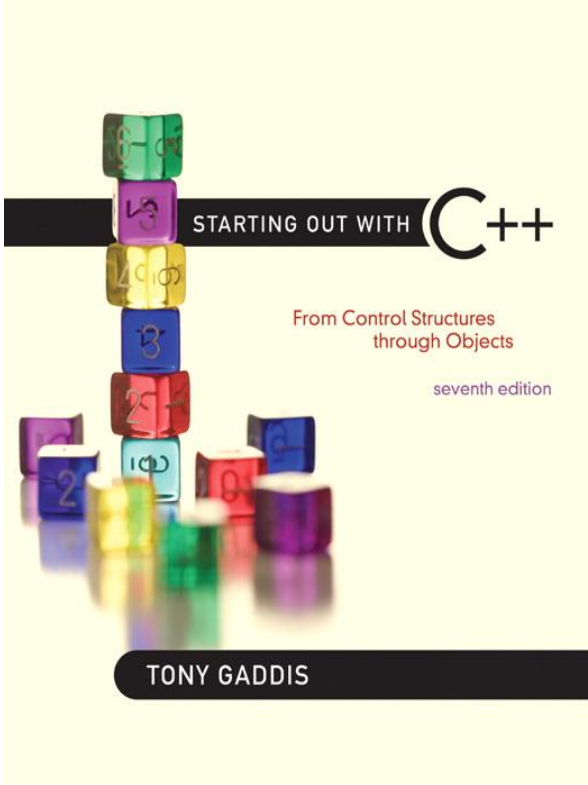
```
Square::Square(int side) :
   Rectangle(side, side)
```

Square inherits from Rectangle

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

Passing Arguments to Base Class Constructor





15.4

Redefining Base Class Functions

Redefining Base Class Functions

- Redefined function:
  - A function in a derived class that has the same name and parameter list as a function in the base class
- Typically used to change the functionality of an inherited function

Redefining Base Class Functions

- Not the same as overloading:
  - with overloading, the parameter lists must be different
- Objects of the base class use the base class version of function
- Objects of the derived class use the 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 Modifies SetScore Function

```
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
```

CurvedActivity Class

- pr 15-7

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 X();// unmodified
void Y();// refedined
```

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

Object `D` invokes function `X()` In `BaseClass`.

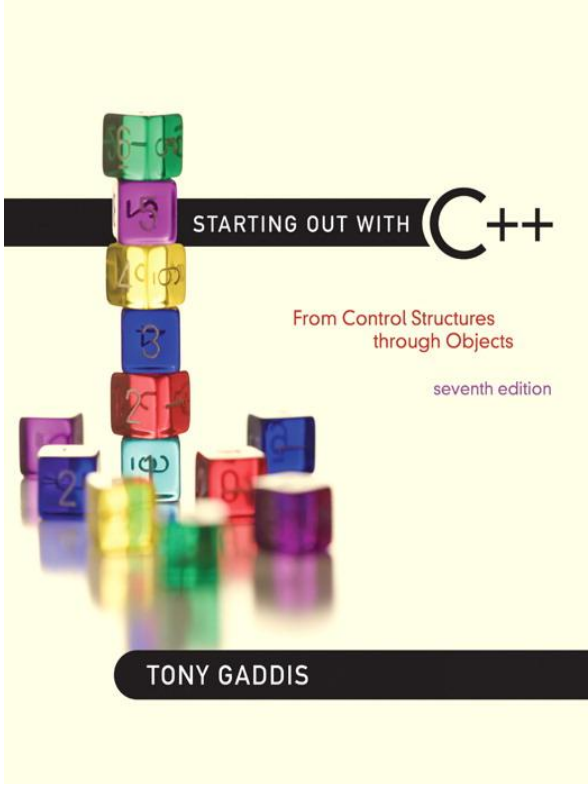
function `X()` invokes function `Y()` in `BaseClass`, not function `Y()` in `DerivedClass`

This is because function calls are bound at compile time.

This is *static binding*.

BaseClass / DerivedClass Classes

- pr 15-8

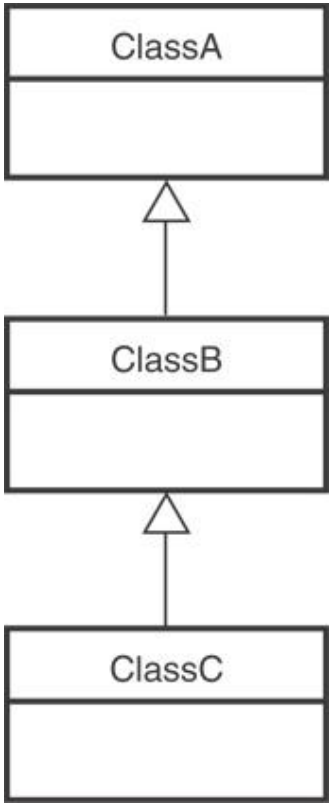


15.5

Class Hierarchies

Class Hierarchies

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

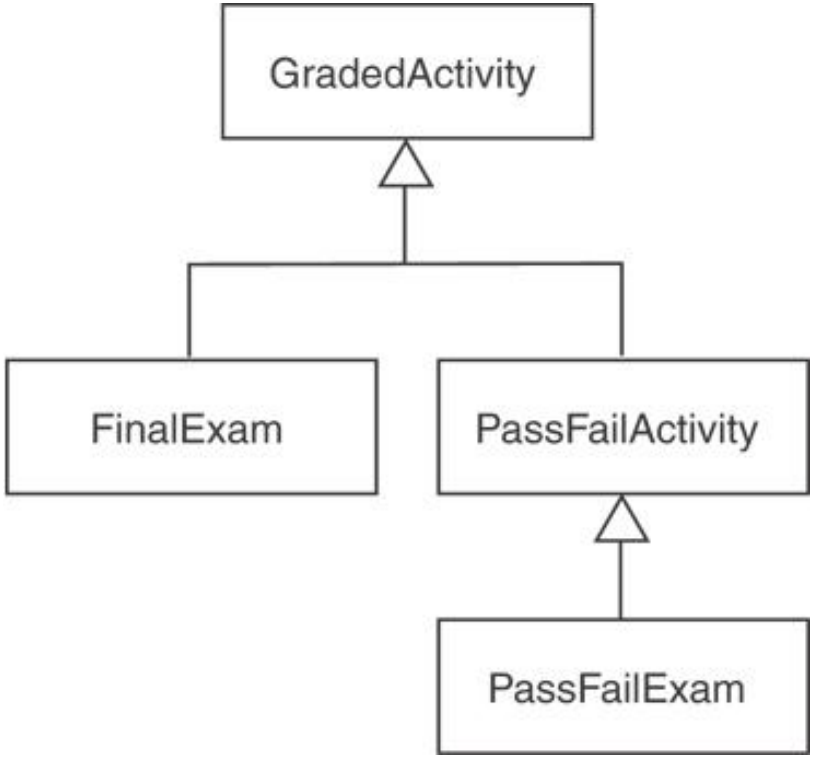


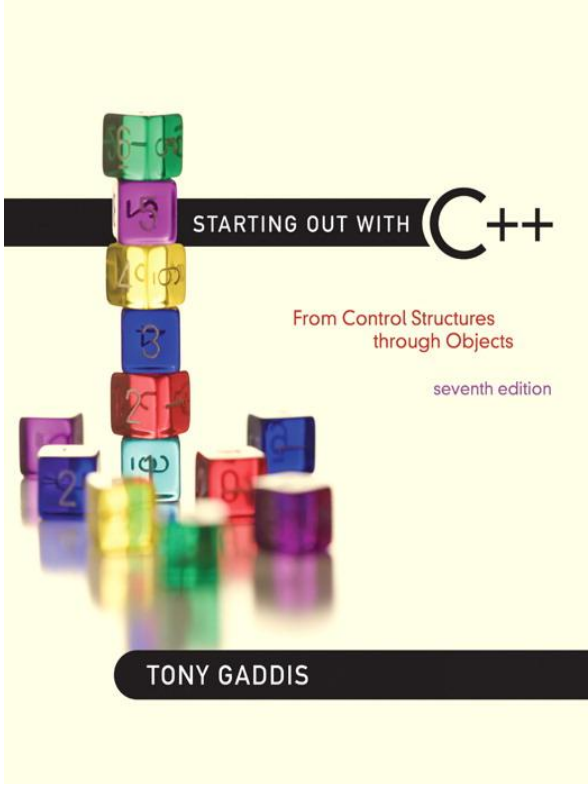
PassFailActivity, PassFailExam

- pr 15-9

Class Hierarchies

- Consider the GradedActivity, FinalExam, PassFailActivity, PassFailExam hierarchy in Chapter 15.





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 the function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Consider this function (from Program 15-9)

```
29  void displayGrade(const GradedActivity &activity)
30  {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33          << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35          << activity.getLetterGrade() << endl;
36  }
```

Because the parameter in the displayGrade function is a GradedActivity reference variable, it can reference any object that is derived from GradedActivity.

That means we can pass a GradedActivity object, a FinalExam object, a PassFailExam object, or any other object that is derived from GradedActivity.

A problem occurs in Program 15-10 however...

Program 15-10

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailActivity.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11    // Create a PassFailActivity object. Minimum passing
12    // score is 70.
13    PassFailActivity test(70);
14
15    // Set the score to 72.
16    test.setScore(72);
17
18    // Display the object's grade data. The letter grade
19    // should be 'P'. What will be displayed?
20    displayGrade(test);
21    return 0;
22 }
```

```
23
24  //*************************************************************
25  // The displayGrade function displays a GradedActivity object's *
26  // numeric score and letter grade.                              *
27  //*************************************************************
28
29  void displayGrade(const GradedActivity &activity)
30  {
31     cout << setprecision(1) << fixed;
32     cout << "The activity's numeric score is "
33          << activity.getScore() << endl;
34     cout << "The activity's letter grade is "
35          << activity.getLetterGrade() << endl;
36  }
```

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is C
```

As you can see from the example output, the `getLetterGrade` member function returned ‘C’ instead of ‘P’.

This is because the GradedActivity class’s `getLetterGrade` function was executed instead of the PassFailActivity class’s version of the function.

Static Binding

- Program 15-10 displays 'C' instead of 'P' because the call to the `getLetterGrade` function is statically bound (at compile time) with the GradedActivity class's version of the function.

- We can remedy this by making the function virtual.

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.

PassFailActivity

- pr 15-10
  - GradedActivity &
  - virtual keyword not used

Virtual Functions

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

```
virtual char getLetterGrade() const;
```

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

GradedActivity version 3

- pr 15-11
  - void displayGrade(const GradedActivity &);
  - virtual char getLetterGrade() const declared in GradedActivity.h
  - function declared virtual in derived class PassFailActivity (not required)

Updated Version of GradedActivity

```
6  class GradedActivity
7  {
8  protected:
9     double score;   // To hold the numeric score
10 public:
11    // Default constructor
12    GradedActivity()
13       { score = 0.0; }
14
15    // Constructor
16    GradedActivity(double s)
17       { score = s; }
18
19    // Mutator function
20    void setScore(double s)
21       { score = s; }
22
23    // Accessor functions
24    double getScore() const
25       { return score; }
26
27    virtual char getLetterGrade() const;
28 };
```

The function is now virtual.

The function also becomes virtual in all derived classes automatically!

If we recompile our program with the updated versions of the classes, we will get the right output, shown here: (See Program 15-11 in the book.)

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is P
```

This type of behavior is known as polymorphism. The term polymorphism means the ability to take many forms.

Program 15-12 demonstrates polymorphism by passing objects of the GradedActivity and PassFailExam classes to the displayGrade function.

Program 15-12

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include "PassFailExam.h"
 4  using namespace std;
 5
 6  // Function prototype
 7  void displayGrade(const GradedActivity &);
 8
 9  int main()
10  {
11     // Create a GradedActivity object. The score is 88.
12     GradedActivity test1(88.0);
13
14     // Create a PassFailExam object. There are 100 questions,
15     // the student missed 25 of them, and the minimum passing
16     // score is 70.
17     PassFailExam test2(100, 25, 70.0);
18
19     // Display the grade data for both objects.
20     cout << "Test 1:\n";
21     displayGrade(test1);    // GradedActivity object
22     cout << "\nTest 2:\n";
```

```
23      displayGrade(test2);    // PassFailExam object
24      return 0;
25   }
26
27   //*************************************************************
28   // The displayGrade function displays a GradedActivity object's *
29   // numeric score and letter grade.                              *
30   //*************************************************************
31
32   void displayGrade(const GradedActivity &activity)
33   {
34      cout << setprecision(1) << fixed;
35      cout << "The activity's numeric score is "
36           << activity.getScore() << endl;
37      cout << "The activity's letter grade is "
38           << activity.getLetterGrade() << endl;
39   }
```

Program Output

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```

Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer, as demonstrated in the `displayGrade` function.

GradedActivity / PassFailExam

- pr 15-12
  - GradedActivity object
  - PassFailExam object
  - virtual char getLetterGrade() const
  - void displayGrade(const GradedActivity &)

Base Class Pointers

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


```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);

cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

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 a derived class will be ignored unless the base class declares the function virtual

PassFailExam / GradedActivity

- pr 15-13
  - GradedActivity *
  - polymorphic behavior

PassFailActivity / PassFailExam

- pr 15-14
  - base class pointer to a derived class object
  - array of objects of type GradedActivity

Redefining vs. Overriding

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

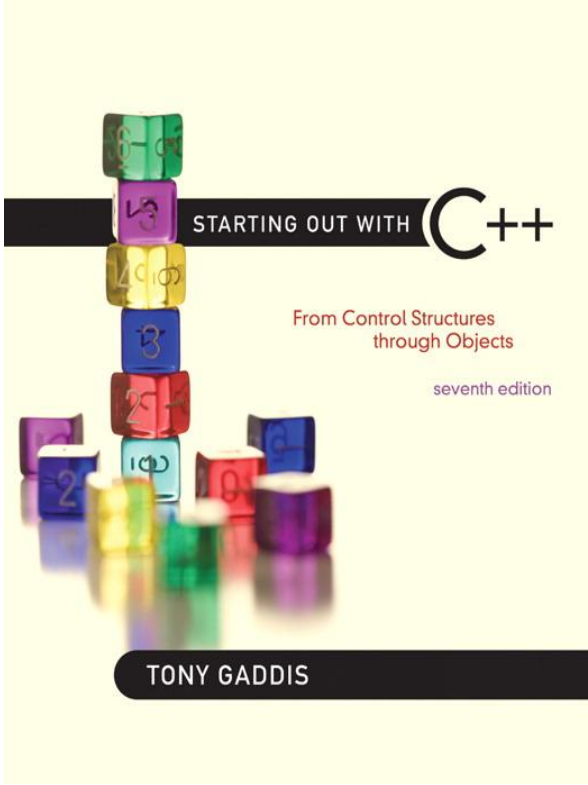
- So, a virtual function is overridden, and a non-virtual function is redefined.

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.
- See Program 15-14 for an example

Animal / Dog Classes

- pr 15-15
  - overrides (redefines a virtual base class function)
- pr 15-16
  - virtual destructor



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
- An abstract base class contains at least one pure virtual function:

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

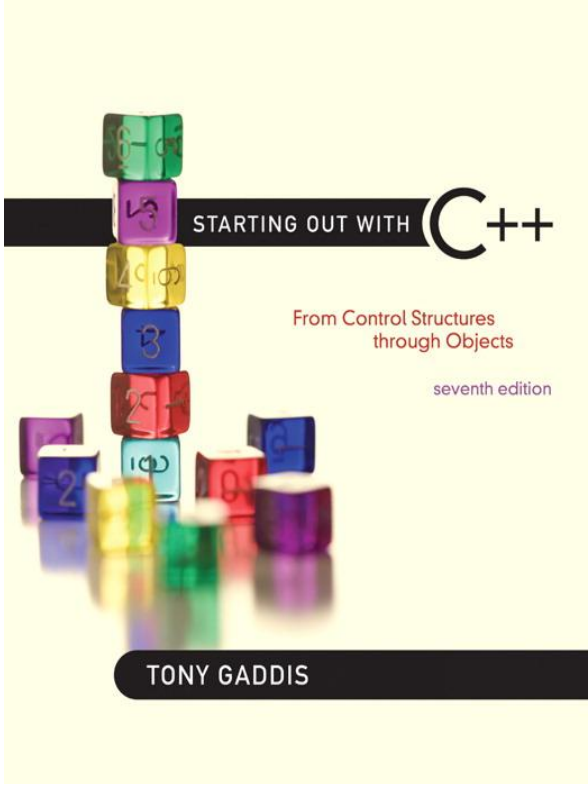
- The `= 0` indicates a pure virtual function
- There can be no function definition in the base class

Abstract Base Classes and Pure Virtual Functions

- Abstract base class:
  - class cannot be used to create 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

CsStudent Class

- pr 15-17
  - abstract class
  - pure virtual function



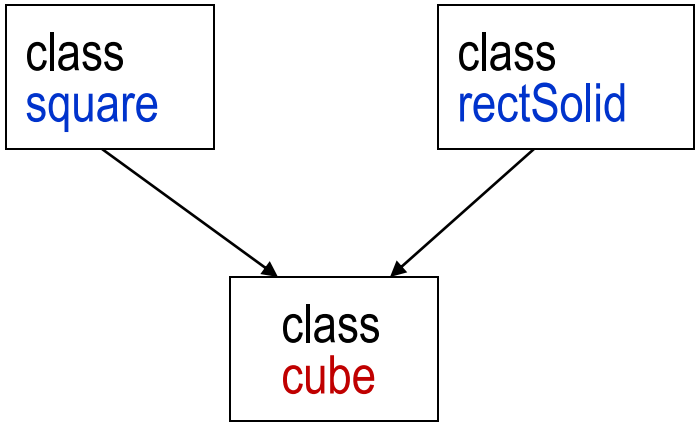
15.8

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 the 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 a member function in a particular base class using the scope resolution operator `::`
- Compiler errors occur if a derived class uses a multiple inheritance base class function without one of these solutions

Date, Time, DateTime Classes

- pr 15-18
  - multiple inheritance