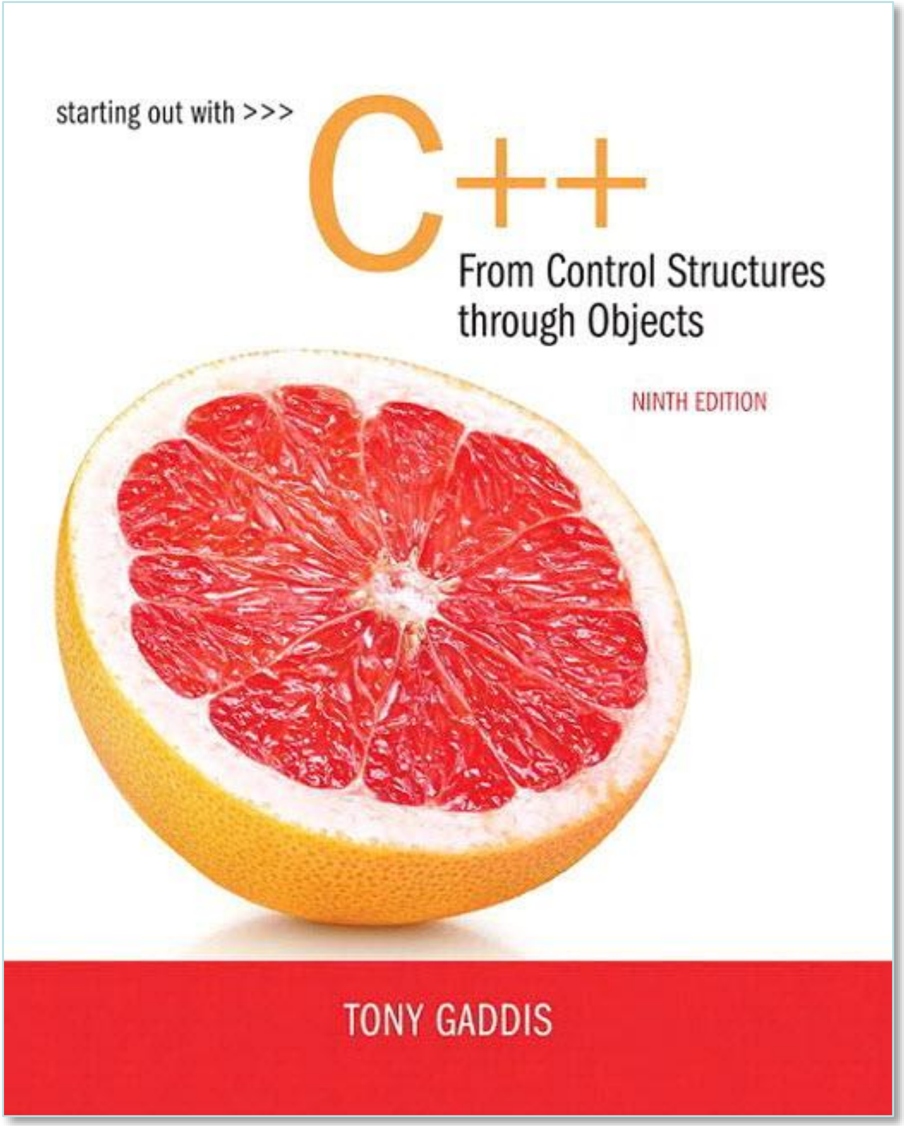
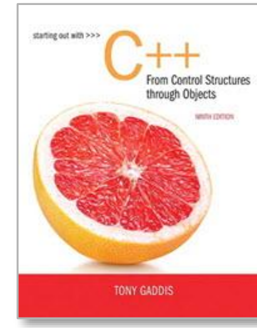
Chapter 15:

Inheritance, Polymorphism, and Virtual Functions





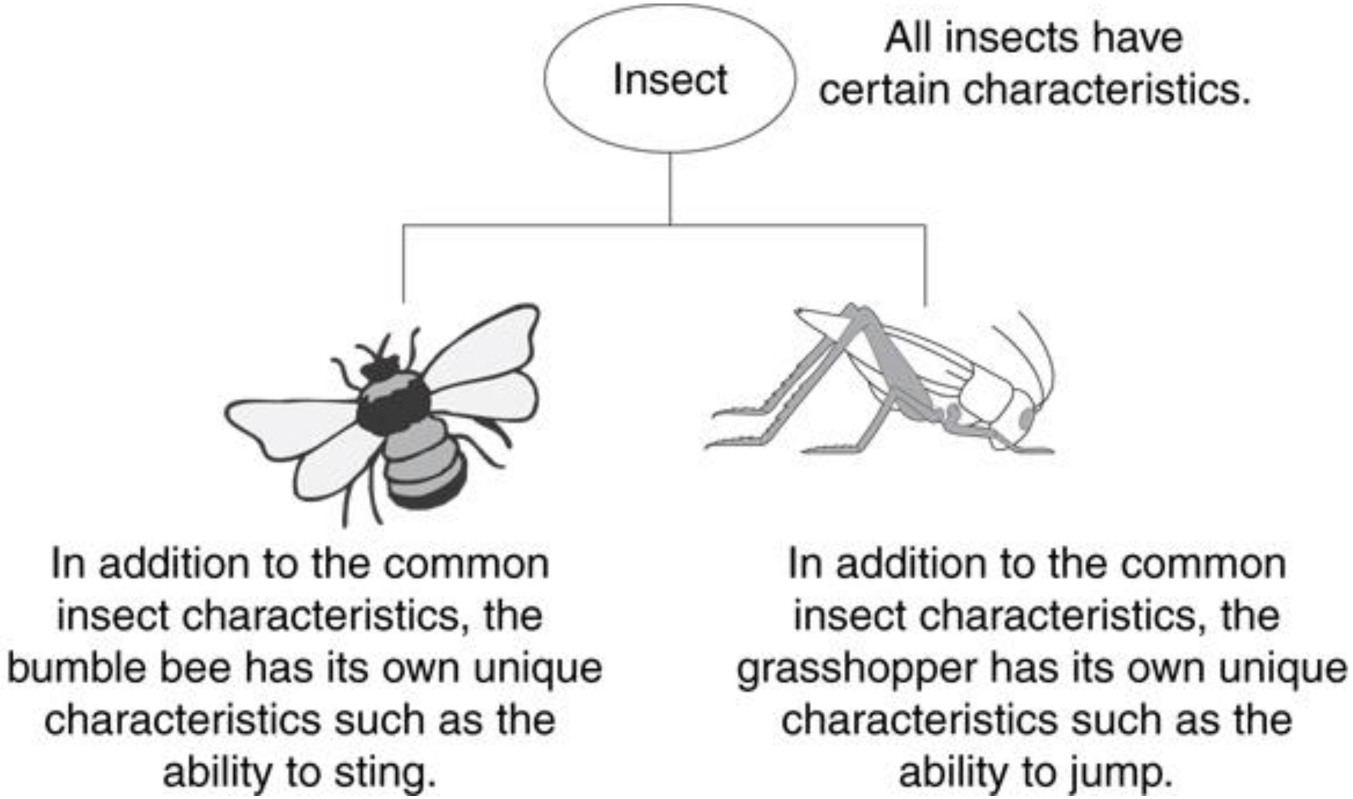
15.1

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

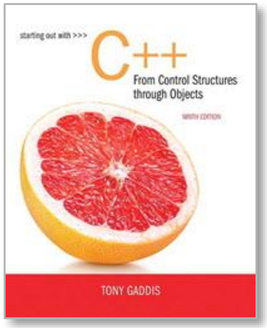
What Does a Child Have?

An object of the derived class has:

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

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

Base class members		How inherited base class members appear in derived class

`private: x`
`protected: y`
`public: z`
	`private` base class →	`x` is inaccessible
		`private: y`
		`private: z`

`private: x`
`protected: y`
`public: z`
	`protected` base class →	`x` is inaccessible
		`protected: y`
		`protected: z`

`private: x`
`protected: y`
`public: z`
	`public` base class →	`x` is inaccessible
		`protected: y`
		`public: z`

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

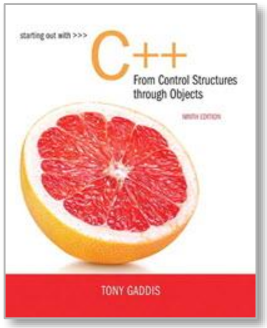
private members:
`char letter;`
`float score;`
`void calcGrade();`
public members:
`void setScore(float);`
`float getScore();`
`char getLetter();`

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

```
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;

//*******************************
// BaseClass declaration       *
//*******************************

```

Program 15-4 *(continued)*

```
class BaseClass
{
public:
   BaseClass()  // Constructor
      { cout << "This is the BaseClass constructor.\n"; }

   ~BaseClass() // Destructor
      { cout << "This is the BaseClass destructor.\n"; }
};

//*******************************
// DerivedClass declaration     *
//*******************************

class DerivedClass : public BaseClass
{
public:
   DerivedClass()  // Constructor
      { cout << "This is the DerivedClass constructor.\n"; }

   ~DerivedClass()  // Destructor
      { cout << "This is the DerivedClass destructor.\n"; }
};

```

Program 5-14 (Continued)

```
//*******************************
// main function               *
//*******************************

int main()
{
   cout << "We will now define a DerivedClass object.\n";

   DerivedClass object;

   cout << "The program is now going to end.\n";
   return 0;
}
```

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

derived class constructor

base class constructor

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

derived constructor parameter

base constructor parameters

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

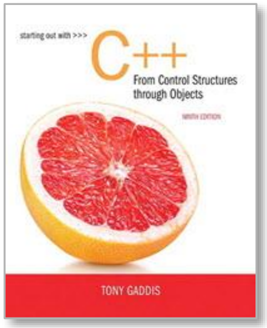
- We can rewrite the MyDerived class as:

```
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

```
#ifndef CURVEDACTIVITY_H
#define CURVEDACTIVITY_H
#include "GradedActivity.h"

class CurvedActivity : public GradedActivity
{
protected:
   double rawScore;     // Unadjusted score
   double percentage;   // Curve percentage
public:
   // Default constructor
   CurvedActivity() : GradedActivity()
      { rawScore = 0.0; percentage = 0.0; }

   // Mutator functions
   void setScore(double s)
      { rawScore = s;
        GradedActivity::setScore(rawScore * percentage); }

   void setPercentage(double c)
      { percentage = c; }

   // Accessor functions
   double getPercentage() const
      { return percentage; }

   double getRawScore() const
      { return rawScore; }
};
#endif
```

Redefined `setScore` function

From Program 15-7

```
// Define a CurvedActivity object.
CurvedActivity exam;

// Get the unadjusted score.
cout << "Enter the student's raw numeric score: ";
cin >> numericScore;

// Get the curve percentage.
cout << "Enter the curve percentage for this student: ";
cin >> percentage;

// Send the values to the exam object.
exam.setPercentage(percentage);
exam.setScore(numericScore);

// Display the grade data.
cout << fixed << setprecision(2);
cout << "The raw score is "
     << exam.getRawScore() << endl;
cout << "The curved score is "
     << exam.getScore() << endl;
cout << "The curved grade is "
     << 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();
```

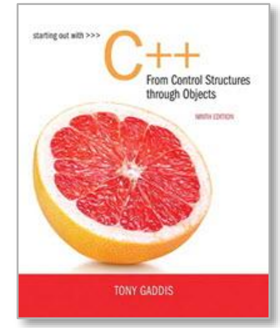
DerivedClass

```
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.

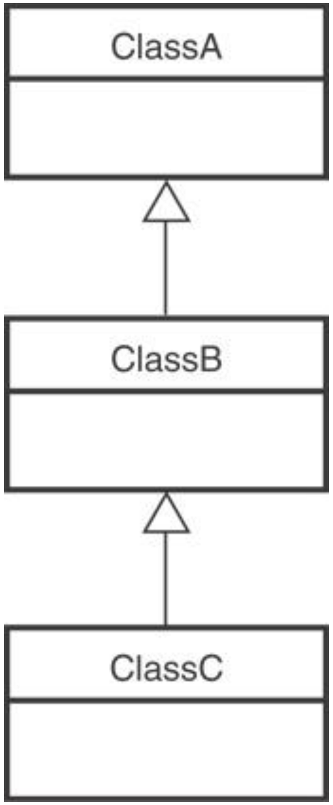


15.5

Class Hierarchies

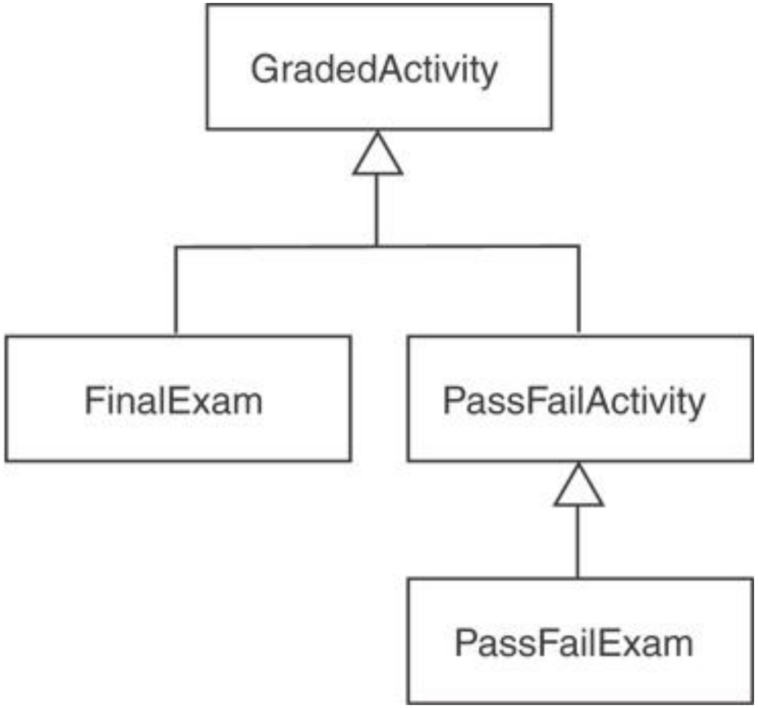
Class Hierarchies

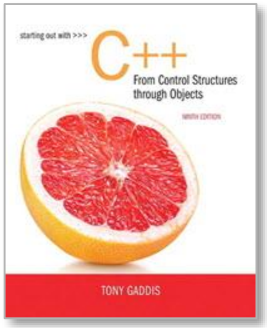
- A base class can be derived from another base class.



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

Consider this function (from Program 15-9)

```
void displayGrade(const GradedActivity &activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

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

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

Program 15-10

```
#include <iostream>
#include <iomanip>
#include "PassFailActivity.h"
using namespace std;

// Function prototype
void displayGrade(const GradedActivity &);

int main()
{
   // Create a PassFailActivity object. Minimum passing
   // score is 70.
   PassFailActivity test(70);

   // Set the score to 72.
   test.setScore(72);

   // Display the object's grade data. The letter grade
   // should be 'P'. What will be displayed?
   displayGrade(test);
   return 0;
}
```

```
//*****************************************************************
// The displayGrade function displays a GradedActivity object's *
// numeric score and letter grade.                               *
//*****************************************************************

void displayGrade(const GradedActivity &activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

Program Output

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

The `getLetterGrade` member function returned 'C' instead of 'P'.

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.

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.

Updated Version of GradedActivity

```
class GradedActivity
{
protected:
   double score;   // To hold the numeric score
public:
   // Default constructor
   GradedActivity()
      { score = 0.0; }

   // Constructor
   GradedActivity(double s)
      { score = s; }

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

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

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

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

```
#include <iostream>
#include <iomanip>
#include "PassFailExam.h"
using namespace std;

// Function prototype
void displayGrade(const GradedActivity &);

int main()
{
   // Create a GradedActivity object. The score is 88.
   GradedActivity test1(88.0);

   // Create a PassFailExam object. There are 100 questions,
   // the student missed 25 of them, and the minimum passing
   // score is 70.
   PassFailExam test2(100, 25, 70.0);

   // Display the grade data for both objects.
   cout << "Test 1:\n";
   displayGrade(test1);    // GradedActivity object
   cout << "\nTest 2:\n";
```

```
    displayGrade(test2);    // PassFailExam object
    return 0;
}

//*****************************************************************
// The displayGrade function displays a GradedActivity object's *
// numeric score and letter grade.                              *
//*****************************************************************

void displayGrade(const GradedActivity &activity)
{
    cout << setprecision(1) << fixed;
    cout << "The activity's numeric score is "
         << activity.getScore() << endl;
    cout << "The activity's letter grade is "
         << activity.getLetterGrade() << endl;
}
```

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.

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 *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.

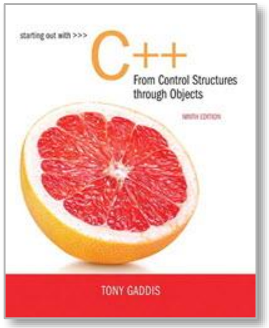
- So, a virtual function is overridden, and a non-virtual function is redefined.

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.
- See Program 15-14 for an example

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.
- See Programs 15-7 and 15-8 for an example



15.7

Abstract Base Classes and Pure Virtual Functions

Abstract Base Classes and Pure Virtual Functions

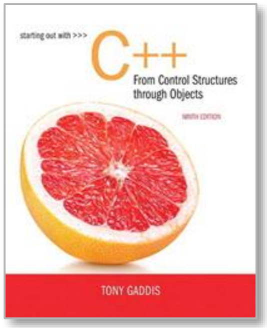
- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:

```
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



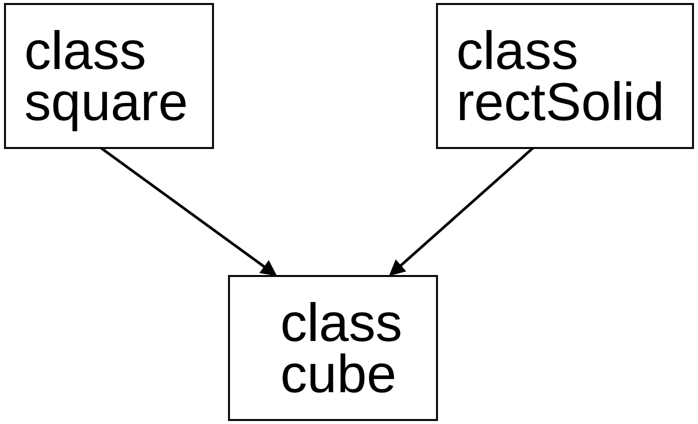
15.8

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