Inheritance, Polymorphism, and Virtual Functions

(Chapter 15. Starting Out with C++: From Control Structures through Objects, Tony Gaddis.)

Le Thanh Huong
School of Information and Communication Technology
Hanoi University of Technology

6.1. 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
  - It inherits all the member variables and functions (except the constructors and destructor) of the class it is based on

Example: Insects

The “is a” relationship

- Inheritance establishes an "is a" relationship between classes.
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete
  - An undergraduate student is a student
Inheritance – Terminology and Notation

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

**Notation:**
```cpp
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 derived object has all of the characteristics of the base class

**Base class** `GradedActivity`

- Create a base class `GradedActivity` that holds the numeric score and provides a method to calculate the equivalent letter grade
- This base class provides a general way of storing and calculating a letter grade for any type of scoring
- For example, quizzes, midterm exams, final exams, essays, lab reports
Inheritance – GradedActivity.h

```c++
#ifndef GRADEDACTIVITY_H
#define GRADEDACTIVITY_H

class GradedActivity
{
private:
    double 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; }

    char getLetterGrade() const;
};
#endif
```

Inheritance – GradedActivity.cpp

```c++
#include "GradedActivity.h"

class GradedActivity
{
private:

public:
    char GradedActivity::getLetterGrade() const
    {
        char letterGrade; // To hold the letter grade
        if (score > 89) letterGrade = 'A';
        else if (score > 79) letterGrade = 'B';
        else if (score > 69) letterGrade = 'C';
        else if (score > 59) letterGrade = 'D';
        else letterGrade = 'F';

        return letterGrade;
    }
};
```

Inheritance – FinalExam.h

```c++
#ifndef FINALEXAM_H
#define FINALEXAM_H
#include "GradedActivity.h"

class FinalExam : public GradedActivity
{
private:
    int numQuestions;    // Number of questions
double pointsEach;   // Points for each question
    int numMissed;       // Number of questions missed

public:
    FinalExam()
    { numQuestions = 0; pointsEach = 0.0; numMissed = 0; }
    FinalExam(int questions, int missed)
    { set(questions, missed); }
    void set(int, int);  // Defined in FinalExam.cpp
double getNumQuestions() const { return numQuestions; }
double getPointsEach() const { return pointsEach; }
int getNumMissed() const { return numMissed; }
};
#endif
```

Base class GradedActivity

- The numeric score for each type of exam can be calculated differently
  - Equal point value for all questions
  - Different point values for each question
  - Extra credit questions
- The class, FinalExam, is a derived class of GradedActivity
  - This is indicated in FinalExam.h by the line of code
  ```c++
class FinalExam : public GradedActivity
```
- FinalExam inherits the instance variable score and all the methods of the class GradedActivity
- Adds on its own methods specifically needed for grading a final exam
- Final Exam is a Graded Activity
#include "FinalExam.h"
void FinalExam::set(int questions, int missed){
    double numericScore; // To hold the numeric score
    // Set the number of questions and number missed.
    numQuestions = questions;
    numMissed = missed;
    // Calculate the points for each question.
    pointsEach = 100.0 / numQuestions;
    // Calculate the numeric score for this exam.
    numericScore = 100.0 - (missed * pointsEach);
    // Call the inherited setScore function to set the numeric score.
    setScore(numericScore);
}

int main()
{
    int questions; // #questions on the exam
    int missed; // #questions missed by the student
    cout << "How many questions are on the final exam? ";
    cin >> questions;
    cout << "How many questions did the student miss? ";
    cin >> missed;
    // Define a FinalExam object and initialize it with the values entered.
    FinalExam test(questions, missed);
    // Display the test results
    cout << "Each question counts " << test.getPointsEach() << " points.\n";
    cout << "Score: " << test.getScore() << endl;
    cout << "Grade: " << test.getLetterGrade() << endl;
    return 0;
}

Inheritance example

- Every FinalExam is a GradedActivity but not every GradedActivity is a Final Exam
- A Quiz may be another derived class of GradedActivity
  - There may be 5 quizzes during a semester, each worth 20 points
- Think of the base class as being contained within the derived class so that FinalExam has access to all of the variables and methods of GradedActivity

Inheritance example

FinalExam consists of the following:
- Private members
  - int numQuestions
  - int numMissed
- double pointsEach
- Public members
  - FinalExam()
  - FinalExam (int, int)
  - set(int, int)
  - getNumQuestions()
  - getNumMissed()
  - setScore(double)
  - getScore()
  - getLetterGrade()
- The variable score from GradedActivity is not listed as a member of the FinalExam class. It is inherited by the derived class, but because it is a private member of the base class, only member functions of the base class may access it.
Getting in “Shape”

Base class `Shape` has an instance variable `area` and functions `getArea` and `setArea`

Derived class `Circle` has its own instance variable `radius` and functions `getRadius` and `setRadius`

- Note that `setRadius` calls function `setArea` and passes the area of the circle using the formula \( \pi r^2 \)
- This sets the value of area in the base class. Since it is private, `Circle` can access it only via the public functions

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

int main()
{
    Circle c;
    c.setRadius(10.0);
    cout<<"Area of the circle is "<<c.getArea()<<endl;
    return 0;
}
```

```
class Shape{
private:
    double area;
public:
    void setArea(double a) {area = a;}
    double getArea() {return area;}
};

class Circle: public Shape
{
private:
    double radius;
bli
    public:
        void setRadius(double r){
            radius = r;
            setArea(3.14*r*r);
        }
    double getRadius() {return radius;}
};
```

Getting in “Shape”

Create a new class named Rectangle. It has

- private instance variables `length` and `width` of type double
- public functions `getLength`, `getWidth` and `setLengthAndWidth`

```
    setLengthAndWidth calls setArea with the value `length*width`
    Add code to the main program to instantiate a Rectangle object r, set the length to 6 and width to 9 and then call the getArea function on r
```
Getting in “Shape”

```cpp
class Rectangle: public Shape
{
    private:
        double length;
        double width;
    public:
        void setLengthAndWidth(double l, double w)
        {
            length = l;
            width = w;
            setArea(length*width);
        }
        double getLength() {return length;}
        double getWidth() {return width;}
};
```

# Getting “Shape”

```cpp
#include <iostream>
#include "Circle.h"
#include "Rectangle.h"
using namespace std;

int main()
{
    Circle c;
    c.setRadius(10.0);
    cout<<"Area of the circle is"<<c.getArea()<<endl;
    Rectangle r;
    r.setLengthAndWidth(6,9);
    cout<<"Area of the rectangle is"<<r.getArea()<<endl;
    return 0;
}
```

Area of the circle is 314
Area of the rectangle is 54

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

6.2. 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

<table>
<thead>
<tr>
<th>Base class members</th>
<th>How inherited base class members appear in derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>private: x</td>
<td>x is inaccessible</td>
</tr>
<tr>
<td>protected: y</td>
<td>private: y</td>
</tr>
<tr>
<td>public: z</td>
<td>protected: z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base class member access in the derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Class Access Specification</td>
</tr>
<tr>
<td>------------------------------------</td>
</tr>
<tr>
<td>Private</td>
</tr>
<tr>
<td>Protected</td>
</tr>
<tr>
<td>Public</td>
</tr>
</tbody>
</table>

- Member access: how a class member (instance variable) is declared within the class (private, protected or public)
- Base Class Access: how inherited base class members are accessed
- The table above gives the relationships
  - An instance variable (count) is declared public in class A
  - Class B is a derived class from A with base class access of protected
  - count is a protected member of the derived class

More Inheritance vs. Access

When Test class inherits from Grade class using public class access, it looks like this:
More Inheritance vs. Access

- **class Grade**
  - private members:
    - char letter;
    - float score;
  - void calcGrade();
  - float getScore();
  - char getLetter();

- **class Test : protected Grade**
  - private members:
    - int numQuestions;
    - float pointsEach;
    - int numMissed;
  - public members:
    - Test(int, int);
    - void setScore(float);
    - float getScore();
    - float getLetter();

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

6.3. 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

```cpp
#include <iostream>
using namespace std;

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

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

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

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

Constructions & Destructors in Base & Derived Classes

- This program demonstrates the order in which base and derived class constructors and destructors are called.
Constructors & Destructors in Base & Derived Classes

```cpp
int main()
{
    DerivedClass object;
    cout << "We will now define a DerivedClass object.\n";
    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.
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:
  ```cpp
  Square::Square(int side) : Rectangle(side, side)
  ```
- Can also be done with inline constructors
- Must be done if base class has no default constructor

---

**Diagram**

```
<table>
<thead>
<tr>
<th>derived class constructor</th>
<th>base class constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square::Square(int side)</td>
<td>Rectangle(side, side)</td>
</tr>
<tr>
<td>derived constructor parameter</td>
<td>base constructor parameters</td>
</tr>
</tbody>
</table>
```
Passing Arguments to Base Class Constructor

```cpp
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle { private:
    double width;
    double length;
    public:
    // Default constructor
    Rectangle() { width = 0.0; length = 0.0; }
    // Constructor #2
    Rectangle(double w, double len) { width = w; length = len; }
    double getWidth() const { return width; }
    double getLength() const { return length; }
    double getArea() const { return width * length; }
};
#endif
```

Cubes not only have length and width (rectangle properties) but height and volume as well. Thus the need to instantiate a Rectangle object when instantiate a Cube object.

```cpp
#ifndef CUBE_H
#define CUBE_H
#include "Rectangle.h"

class Cube : public Rectangle { protected:
    double height;
    double volume;
    public:
    // Default constructor
    Cube() : Rectangle() { height = 0.0; volume = 0.0; }
    // Constructor #2
    Cube(double w, double len, double h) : Rectangle(w, len)
    { height = h; volume = getArea() * h; }
    double getHeight() const { return height; }
    double getVolume() const { return volume; }
};
#endif
```

6.4. 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
- 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

Class Cube: public Rectangle

```cpp
    …
};
```

Cube: Cube(double w, double len, double h): Rectangle(w, len)

```cpp
    …
};
```
Base Class

```cpp
class GradedActivity {
private:
    char letter; // To hold the letter grade
    double score; // To hold the numeric score

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; }
};
```

Derived Class

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

class CurvedActivity : public GradedActivity {
private:
    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;
        determineGrade();
        GradedActivity::setScore(rawScore * percentage);
    }
    void setPercentage(double c) {
        percentage = c;
    }

    // Accessor functions
    double getPercentage() const { return percentage; }
    double getRawScore() const { return rawScore; }
};
#endif
```

From Program 15-6

```cpp```

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?

---

Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87
The raw score is 87.00
The curved score is 92.22
The curved grade is A
```
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, not function Y() in DerivedClass, because function calls are bound at compile time.
- This is static binding.

Class Hierarchies

6.5. Class Hierarchies

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

Class Hierarchies

PassFailActivity inherits score from GradedActivity and has its own member variable minPassingScore
PassFailExam inherits score and minPassingScore from PassFailActivity and has its own member variables numQuestions, pointsEach, numMissed
PassFailActivity redefines function getLetterGrade so that the letter grade is either P or F and not A,B,C or D as computed by getLetterGrade in GradedActivity

GradedActivity .h and .cpp

#include "GradedActivity.h"

char GradedActivity:::
getLetterGrade() const
{
    char letterGrade;
    if (score > 89)
        letterGrade = 'A';
    else if (score > 79)
        letterGrade = 'B';
    else if (score > 69)
        letterGrade = 'C';
    else if (score > 59)
        letterGrade = 'D';
    else
        letterGrade = 'F';
    return letterGrade;
}

PassFailActivity
GradedActivity
FinalExam
PassFailActivity
PassFailExam

PassFailExam
GradedActivity
6.6. Polymorphism and Virtual Member Functions

- Polymorphism allows an object reference variable or an object pointer to reference objects of different types, and to call the correct member functions, depending upon the type of object being referenced.
Polymorphism and Virtual Member Functions

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

```cpp
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`. 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 this program however...

Program 15-9

```cpp
#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;
}
```

Program 15-9 (cont.)

```cpp
//******************************************************************************
// 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

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 15-9 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.
  - Thus, the actual letter grade is computed instead of the P/F grade.

- We can remedy this by making the `getLetterGrade` 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.
- To make a function virtual, place the virtual keyword before the return type in the base class's declaration:
  
  ```cpp
  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

```cpp
class GradedActivity
{
  protected:
    double score;
  public:
    GradedActivity()
    {
      score = 0.0;
    }
    GradedActivity(double s)
    {
      score = s;
    }

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

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

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

If we recompile our program with the updated versions of the classes, we will get the right output, shown here: (See Program 15-10 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 15-11 demonstrates polymorphism by passing objects of the `GradedActivity` and `PassFailExam` classes to the `displayGrade` function.
Program 15-11
#include "PassFailExam.h"
// Function prototype
void displayGrade(const GradedActivity &); int main()
{ // Create a GradedActivity object. The score is 88.0.
  GradedActivity test1(88.0);
  // Create a PassFailExam object. There are 100 questions, the
  // student missed 25 of them, the minimum passing score is 70.
  PassFailExam test2(100, 25, 70.0);
  // Display the grade data for both objects.
  cout << "Test 1:
"; displayGrade(test1);    // GradedActivity object
  cout << "Test 2:
"; displayGrade(test2);    // PassFailExam object
  return 0;
}

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;
}

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.
- Polymorphic behavior is not possible when an object is passed by value.
- Static binding will take place when the object is not a reference variable or pointer, for example:

```
void displayGrade (const GradedActivity *activity)
```

Base Class Pointers

- Can define a pointer to a base class object
  - Can assign it the address of a derived class object
  - This statement dynamically allocates a PassFailExam object and assigns its address to exam, which is a GradedActivity pointer

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

cout<<exam->getScore()<<endl;
cout<<exam->getLetterGrade()<<endl;
```
# Base Class Pointers

```cpp
#include "PassFailExam.h"

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;
}

int main()
{
    const int NUM_TESTS = 4;
    GradedActivity *tests[NUM_TESTS] = {
        new GradedActivity(88.0),
        new PassFailExam(100, 25, 70.0),
        new GradedActivity(67.0),
        new PassFailExam(50, 12, 60.0)};
    // Display the grade data for each element in the array.
    for (int count = 0; count < NUM_TESTS; count++)
    {
        cout << "Test #" << (count + 1) << endl;
        displayGrade(tests[count]);
    }
    return 0;
}
```

---

**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
Test #3:
The activity’s numeric score is 67.0
The activity’s letter grade is D
Test #4:
The activity’s numeric score is 76.0
The activity’s letter grade is P
```

---

## Base Class Pointers

- The "is-a" relationship does not work in reverse
  - "A final exam is a graded activity" is true but "a graded activity is a final exam" is not true.
  - Not all graded activities are final exams
    - GradedActivity *gaPointer = new GradedActivity(88.0);
    - FinalExam *fePointer = gaPointer; // WILL NOT WORK
  - A final exam has more functionality than a graded activity so you cannot assign `gaPointer` to `fePointer`
  - You can make the assignment with casting; this means `fePointer` points to a base class object.
    - `fePointer = static_cast<FinalExam*>(gaPointer);`
    - `fePointer->getPointsEach()` will compile but a runtime error will occur.

---

## Redefining vs. Overriding

- Redefined functions in derived class will be ignored unless base class declares the function virtual
  - 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.
- If the derived class is pointed to by an object of the base class, only the base class destructor will be called.
  - Making the base class destructor virtual will enable both destructors to execute.
  - When a base class function is declared virtual, all overridden versions of the function in derived classes automatically become virtual.
  - Including a virtual destructor in a base class, even one that does nothing, will ensure that any derived class destructors will also be virtual.

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.
  - The abstract class represents the generic or abstract form of all classes that are derived from it.
- 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
- In Student.h we have a pure virtual function:
  virtual int getRemainingHours() const = 0;
- This is defined in CsStudent.cpp:
  int CsStudent::getRemainingHours() const
  {
    int reqHours, // Total required hours
    remainingHours; // Remaining hours
    // Calculate the required hours.
    reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;
    // Calculate the remaining hours.
    remainingHours = reqHours - (mathHours + csHours + genEdHours);
    // Return the remaining hours.
    return remainingHours;
  }
Abstract Base Classes and Pure Virtual Functions

- There is no meaning to calculating the remaining hours for a general student – each major has its own requirements.
- Thus, the pure virtual function in class `Student` which is given specific definition each derived class.

6.8. 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;
```

- Arguments can be passed to both base classes’ constructors:

```
cube::cube(int side) : square(side),
                     rectSolid(side, side, side);
DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :
            Date(dy, mon, yr),
            Time(hr, mt, sc)
```

- The order the base class constructor calls appear in the list does not matter. They are called in order of inheritance.
- Base class constructors are called in order they are listed in the first line of the class declaration.

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.