C++ Object-Oriented Prog.
Unit 5: Object-Oriented Design

CHAPTER 17: INHERITANCE AND POLYMORPHISM

DR. ERIC CHOU IEEE SENIOR MEMBER

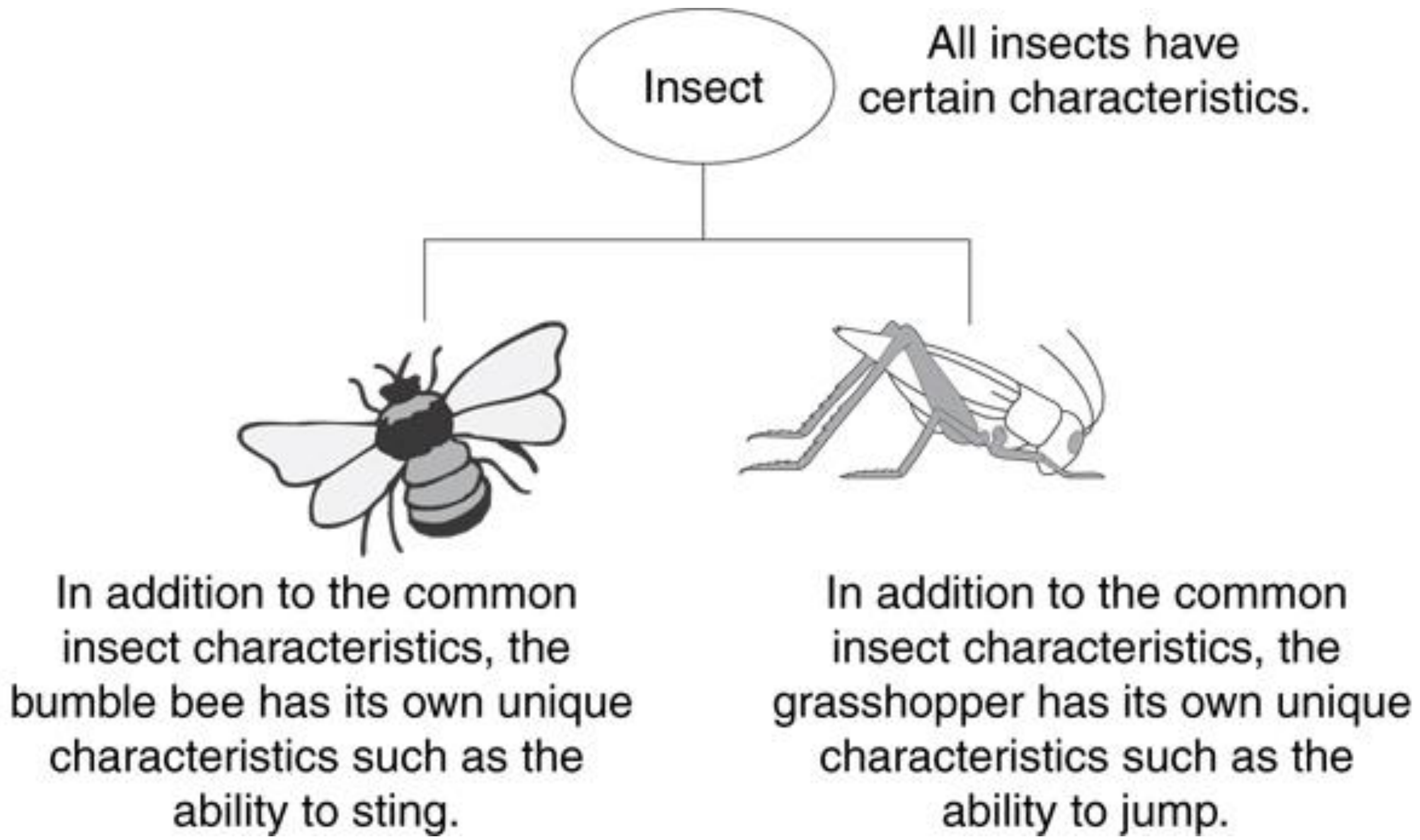
LECTURE 1

Overview

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: Insect Taxonomy



Insect

All insects have certain characteristics.

In addition to the common insect characteristics, the bumble bee has its own unique characteristics such as the ability to sting.

In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

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 in C++

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

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.

Relationship between Base Classes and Derived Classes

Using protected data members

Advantages

- Derived classes can modify values directly
- Slight increase in performance
  - Avoid set/get function call overhead

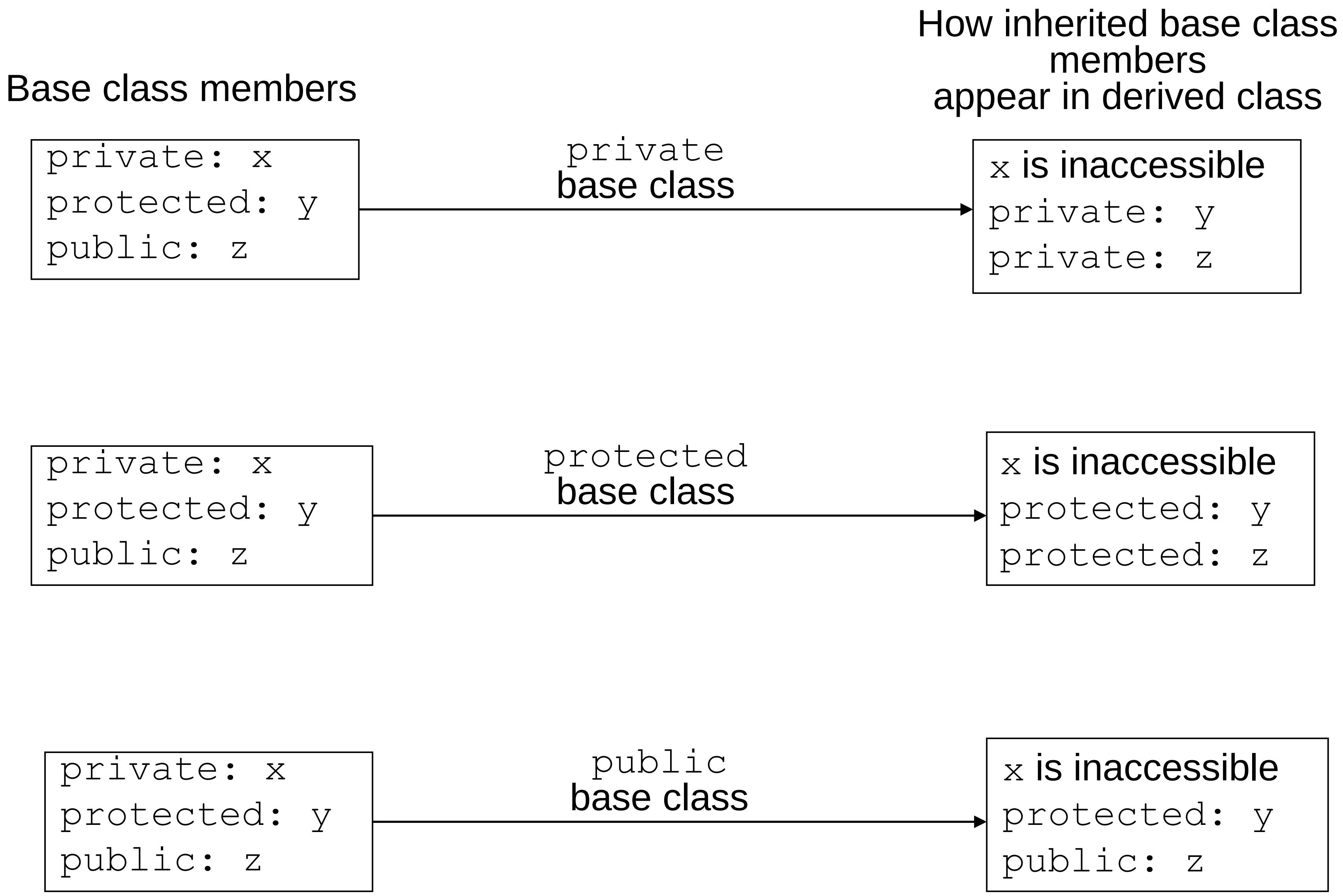
Disadvantages

- No validity checking
  - Derived class can assign illegal value
- Implementation dependent
  - Derived class member functions more likely dependent on base class implementation
  - Base class implementation changes may result in derived class modifications
    - Fragile (brittle) software

LECTURE 2

Is_A and has_A Relationship

Inheritance vs. Access



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();
  char getLetter();
```

Inheritance vs. Access

```
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();
```

Inheritance vs. Access

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);
```

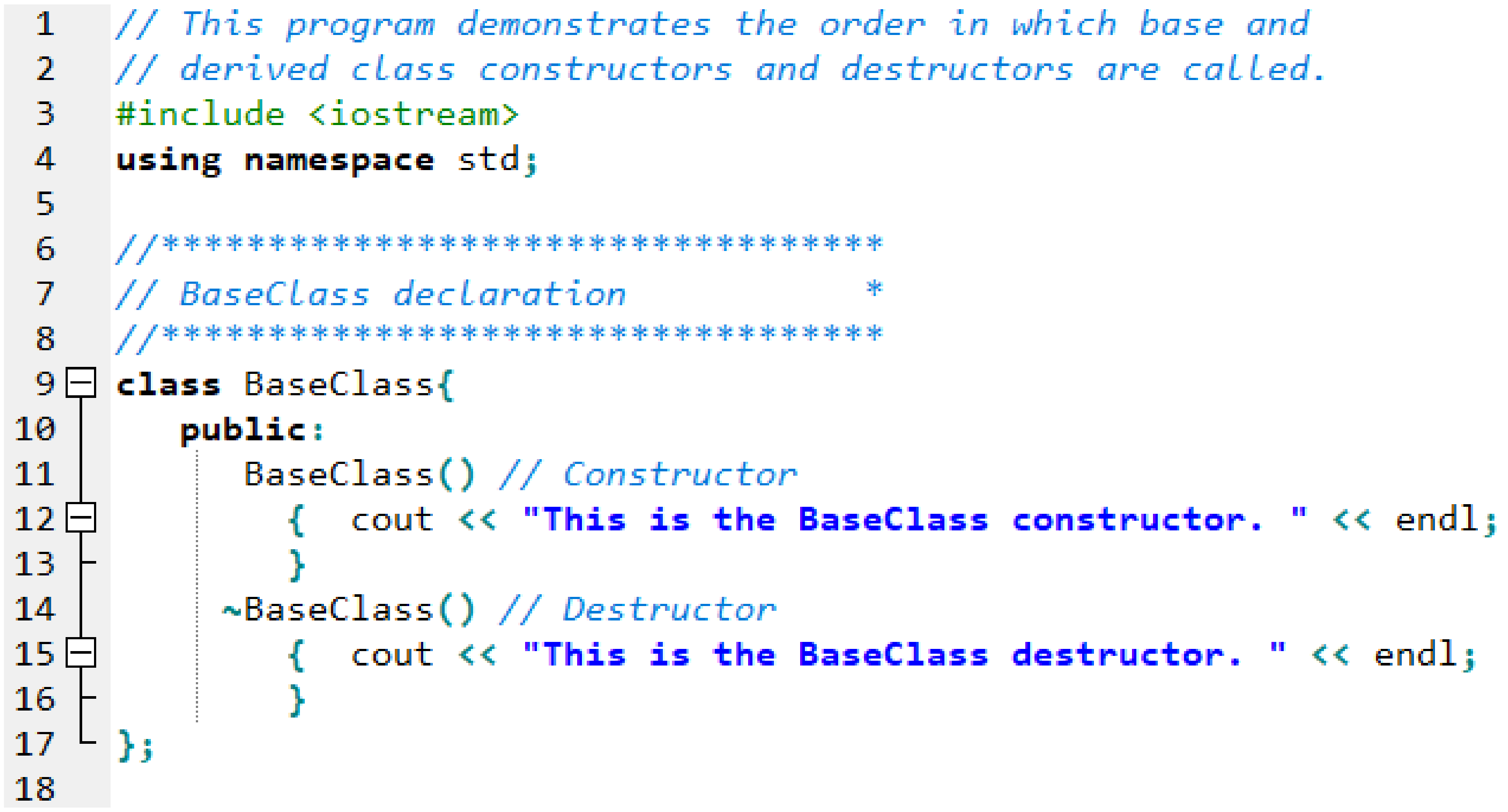
LECTURE 3

Base Class and Derived Class

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

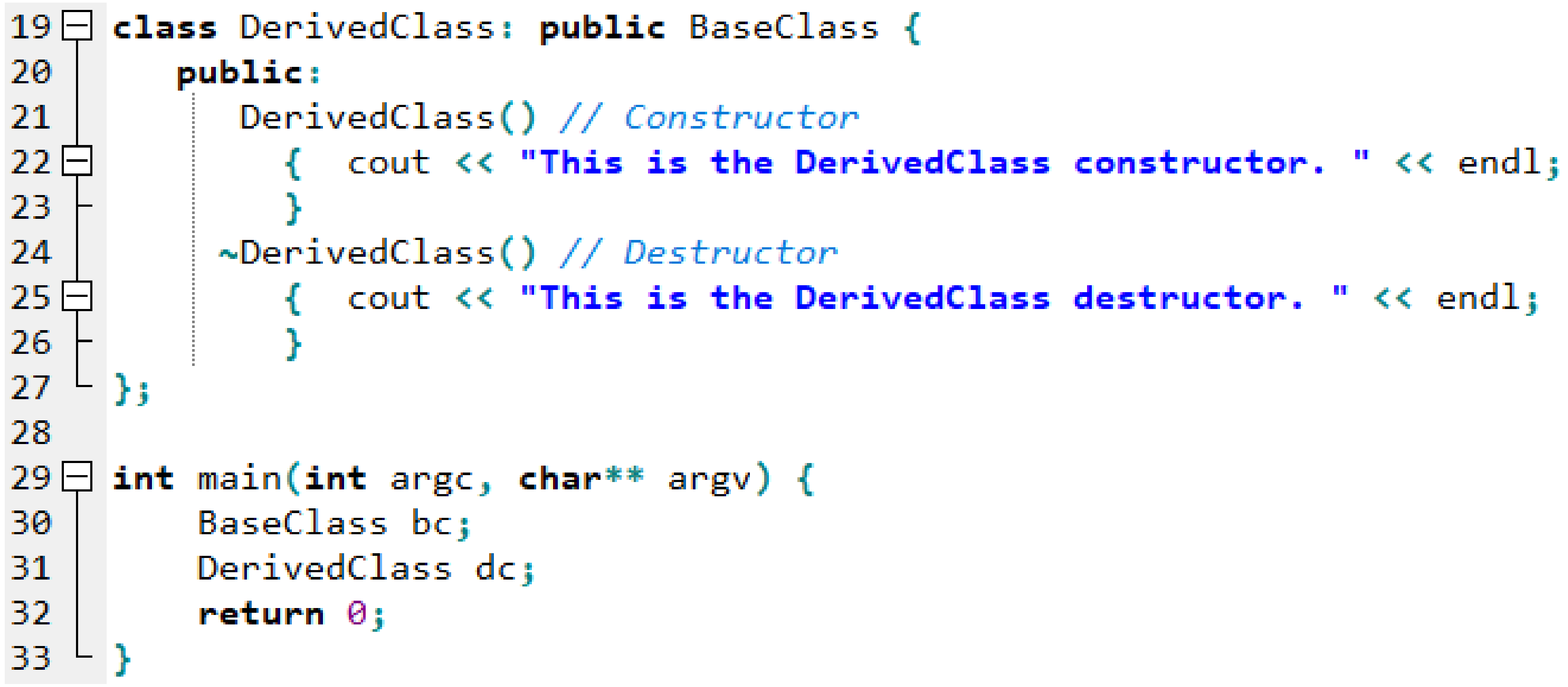


```cpp
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;

//**********************************
// BaseClass declaration            *
//**********************************
class BaseClass{
   public:
      BaseClass() // Constructor
        {  cout << "This is the BaseClass constructor. " << endl;
        }
     ~BaseClass() // Destructor
        {  cout << "This is the BaseClass destructor. " << endl;
        }
};

```

Constructors and Destructors in Base and Derived Classes

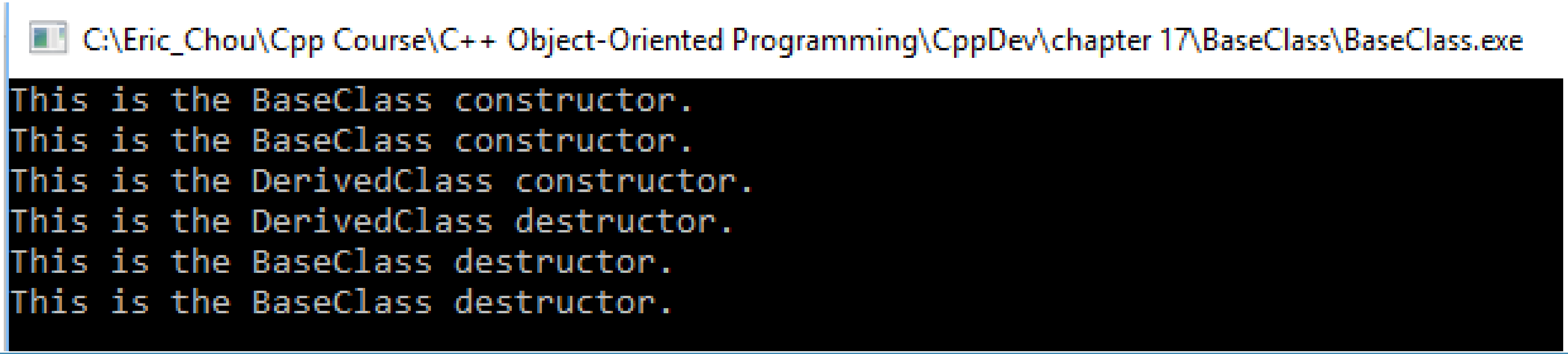


```
class DerivedClass: public BaseClass {
    public:
        DerivedClass() // Constructor
            {   cout << "This is the DerivedClass constructor. " << endl;
            }
       ~DerivedClass() // Destructor
            {   cout << "This is the DerivedClass destructor. " << endl;
            }
};

int main(int argc, char** argv) {
        BaseClass bc;
        DerivedClass dc;
        return 0;
}
```

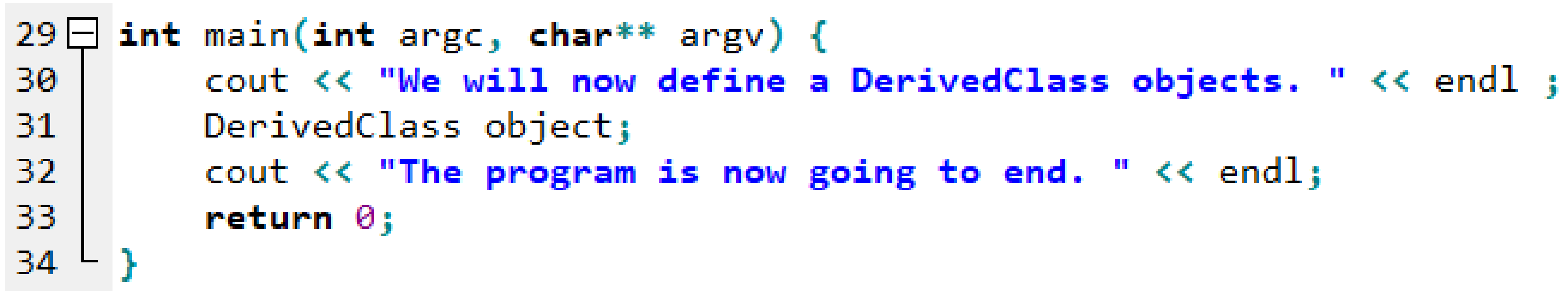
Demo Program: BaseClass.cpp

Go Dev C++!!!


C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 17\BaseClass\BaseClass.exe

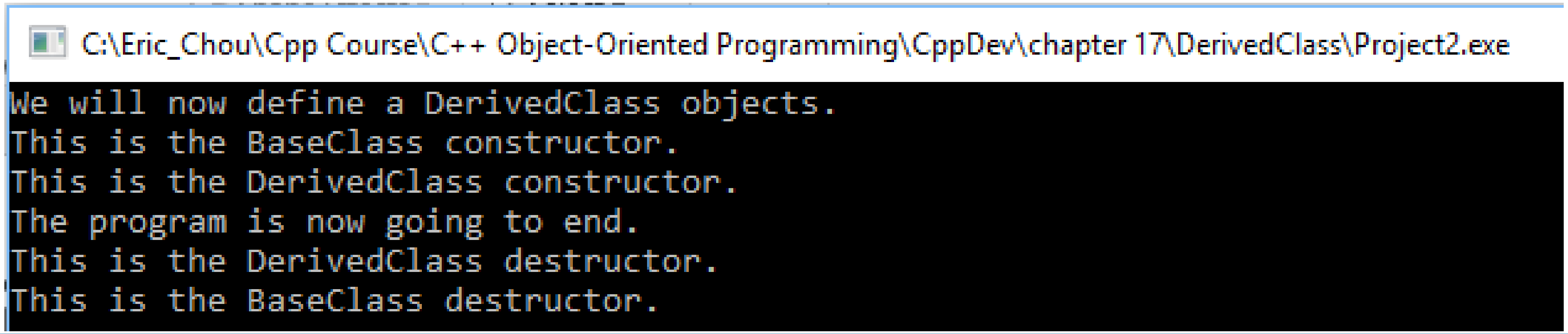
```
This is the BaseClass constructor.
This is the BaseClass constructor.
This is the DerivedClass constructor.
This is the DerivedClass destructor.
This is the BaseClass destructor.
This is the BaseClass destructor.
```

Constructors and Destructors in Base and Derived Classes



```
int main(int argc, char** argv) {
	cout << "We will now define a DerivedClass objects. " << endl ;
	DerivedClass object;
	cout << "The program is now going to end. " << endl;
	return 0;
}
```

Demo Program: DerivedClass.cpp



C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 17\DerivedClass\Project2.exe

```
We will now define a DerivedClass objects.
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.
```

LECTURE 4

Passing Arguments to Base Class

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

$$\overbrace{\texttt{Square::Square(}\underbrace{\texttt{int side}}_{\text{derived constructor parameter}}\texttt{)}}^{\text{derived class constructor}}\texttt{:}\overbrace{\texttt{Rectangle(}\underbrace{\texttt{side,side}}_{\text{base constructor parameters}}\texttt{)}}^{\text{base class constructor}}$$

LECTURE 5

Redefining

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; }
};
```

Derived Class

```
#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; }
};
```

Redefined `setScore` function

Driver Program

```cpp
   // 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;
```

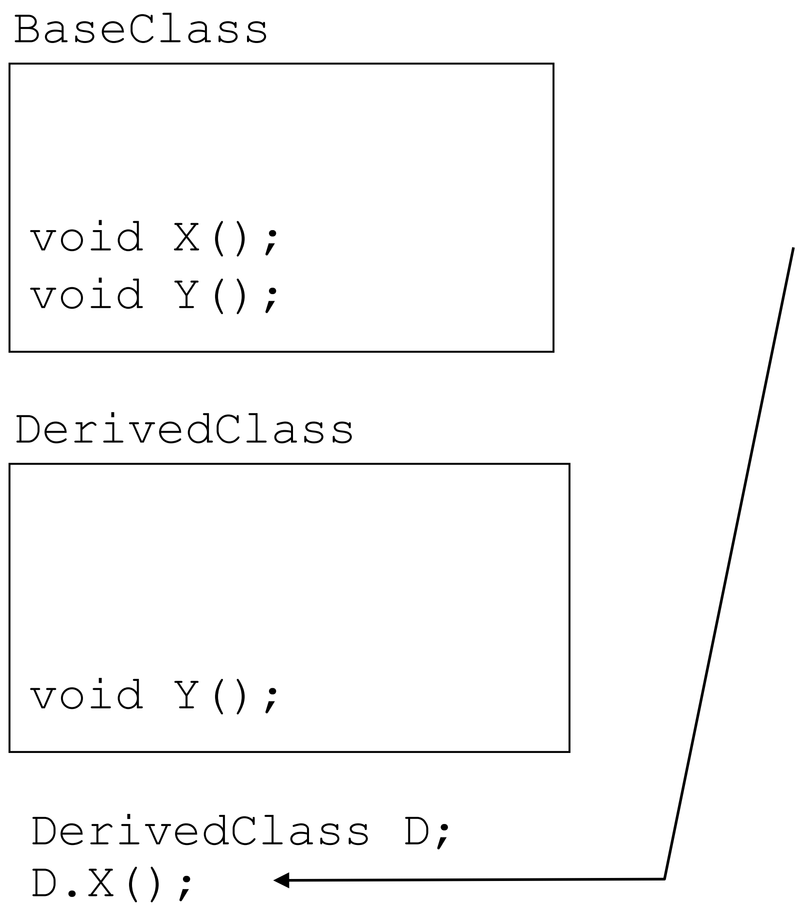
Program Output with Example Input Shown in Bold

```
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();
```

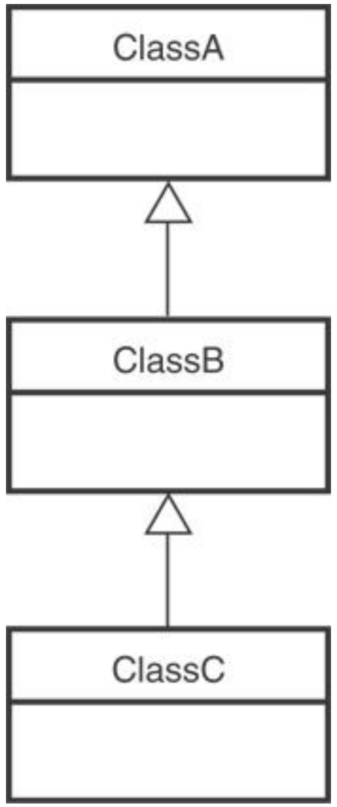
Object `D` invokes function `X()` In `BaseClass`. Function `X()` invokes function `Y()` in `BaseClass`, not function `Y()` in `DerivedClass`, because function calls are bound at compile time. This is static binding.

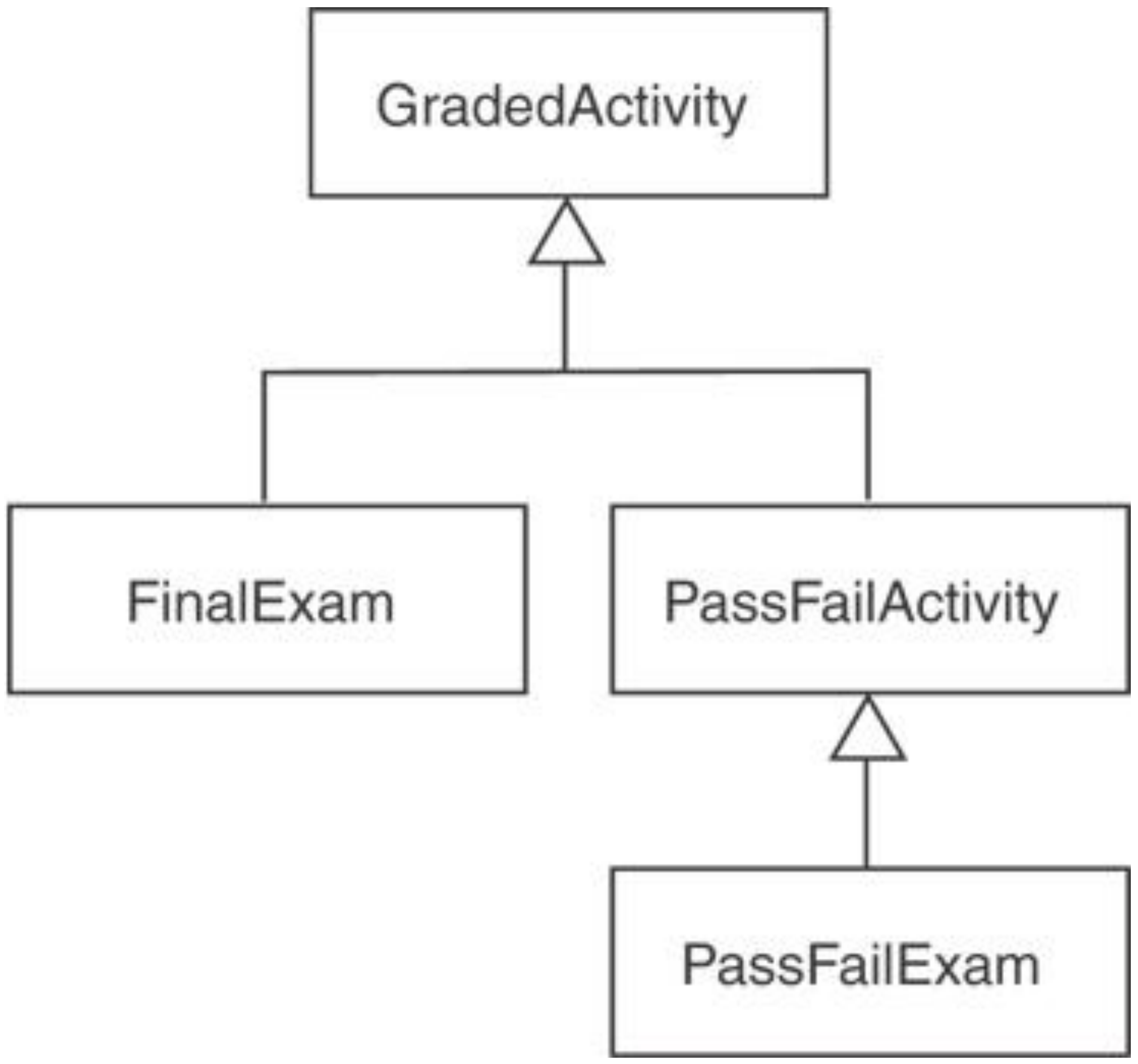
LECTURE 6

Class Hierarchy

Class Hierarchies

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





Class Hierarchies

- Consider the GradedActivity, FinalExam, PassFailActivity, PassFailExam hierarchy.

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**

Polymorphism and Virtual Member Functions

```
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 Program 15-10 however...

Program 15-10

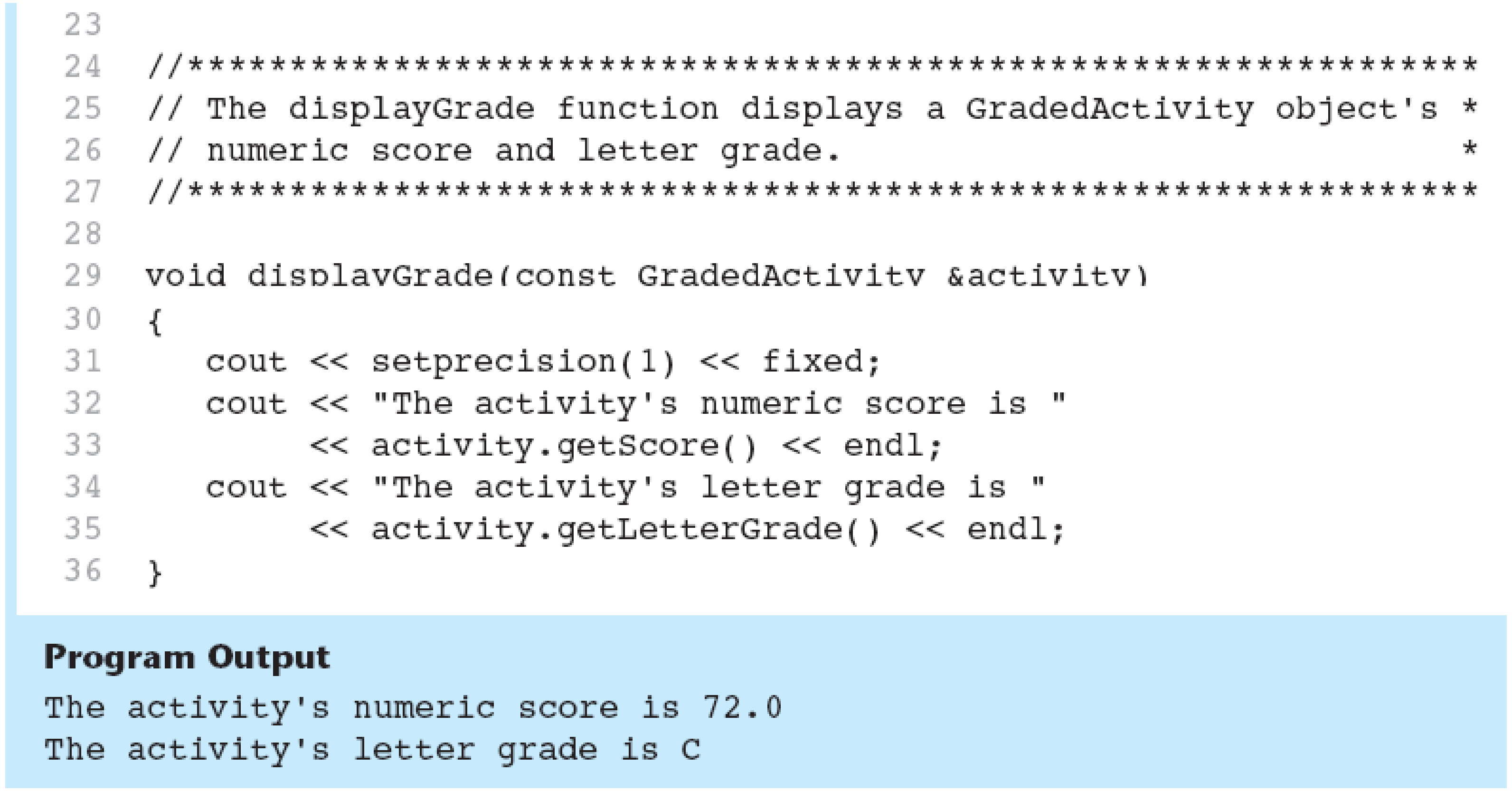
```
#include <iostream>
#include <iomanip>
#include "PassFailActivity.h"
using namespace std;

// 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
```

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.

LECTURE 7

Static Binding and Dynamic Binding

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!

Polymorphism

- 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`

LECTURE 8

Overloading Redefining and Overriding

Overloading, Redefining, and Overriding

Overloading: applies to same function name but different function signature. For example, the overloading of sort algorithm for different subjects to be sorted.

Redefining: applies to different physical functions of same function signature.

Overriding: applies to different virtual functions of same function signature.

Generic programming (template): class type definition to allow the class to be container of different data types. Or functional definition to allow the function to processing different data types of the same functionality.

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

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 method)
- 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

Interface

- C++ language has no interface.
- C++ abstract classes without data fields and with only pure virtual methods (abstract method) are equivalent to interface in other languages.

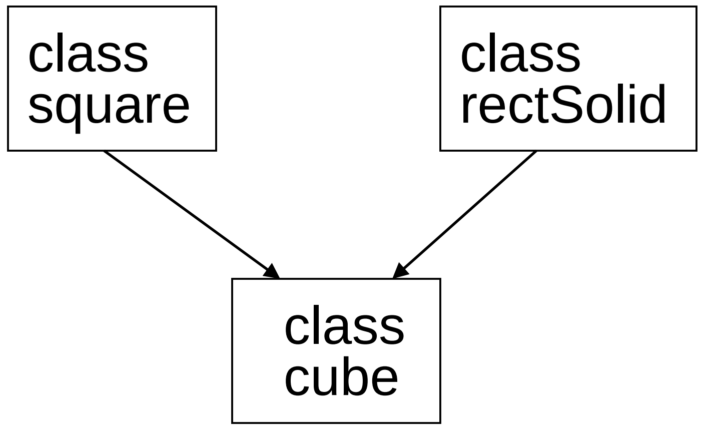
LECTURE 9

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;
```



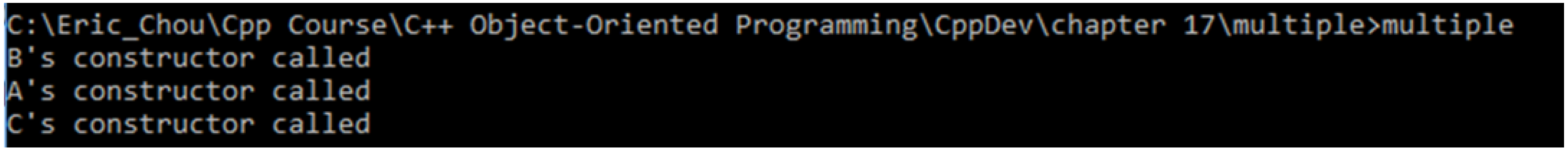
Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
  1. Derived class redefines the multiply-defined function
  2. 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

Multiple Inheritance in C++

Demo Program: multiple.cpp

- Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.
- The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.



```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 17\multiple>multiple
B's constructor called
A's constructor called
C's constructor called
```

Execution of Constructors in Multiple Inheritance in C++

Method of Inheritance	Order of Execution

class B: public A{
};
	A(): base constructor
	B(): derived constructor

class C: public B, public A{
};
	B(): base constructor
	A(): base constructor
	C(): derived constructor

class C: public B, virtual A{
};
	A(): virtual base constructor
	B(): base constructor
	C(): derived constructor

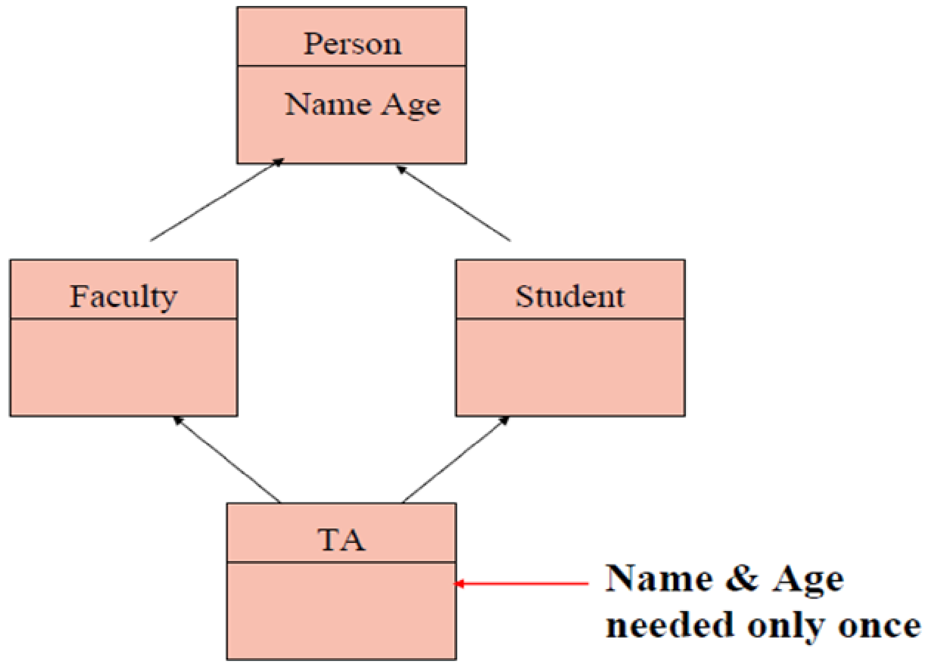
class B: public A{
};
class C: public B{
};
	A(): super base constructor
	B(): base constructor
	C(): derived constructor

LECTURE 10

Inheritance Diamond Problem (I): Problem and Solution

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



Demo Program (I): diamond0.cpp

```cpp
#include<iostream>
using namespace std;
class Person {
   // Data members of person
public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
};

class Faculty : public Person {
   // data members of Faculty
public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
```

Demo Program (II): diamond0.cpp

```cpp
class Student : public Person {
   // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};
class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x)   {
        cout<<"TA::TA(int ) called"<< endl;
    }
};
int main()  {
    TA ta1(30);
}
```

Execution Results

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

- In the above program, constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed.

- So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities.

Solution 1:

The solution to this problem is ‘**virtual**’ keyword. We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class.

For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
    Person()     { cout << "Person::Person() called" << endl;   }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
```

```
class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x)   {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main()  {
    TA ta1(30);
}
```

Demo Program: diamond1.cpp

Go Notepad++!!!

Execution Results

```
Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

- In the above program, constructor of ‘Person’ is called once. One important thing to note in the above output is, the default constructor of ‘Person’ is called.
- When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

LECTURE 11

Inheritance Diamond Problem (II): Passing Parameters

How to call the parameterized constructor of the ‘Person’ class?

- The constructor has to be called in ‘TA’ class.
- For example, see the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x)  { cout << "Person::Person(int ) called" << endl; }
    Person()     { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x)   {
       cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
```

```
class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};
class TA : public Faculty, public Student  {
public:
    TA(int x):Student(x), Faculty(x), Person(x)   {
        cout<<"TA::TA(int ) called"<< endl;
    }
};
int main()  {
    TA ta1(30);
}
```

Execution Results:

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

- In general, it is not allowed to call the grandparent’s constructor directly, it has to be called through parent class.
- It is allowed only when ‘virtual’ keyword is used.

LECTURE 12

C++ Friend Functions

Friendship and inheritance

- In principle, **private** and **protected** members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends".
- Friends are functions or classes declared with the **friend** keyword.
- A **non-member** function can access the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword **friend.** (Declaration of a non-member fucction or class.)

Demo Program: friend_function.cpp

- The duplicate function is a friend of class Rectangle. Therefore, function duplicate is able to access the members width and height (which are private) of different objects of type Rectangle.
- Notice though that neither in the declaration of duplicate nor in its later use in main, function duplicate is considered a member of class Rectangle. It isn't! It simply has access to its private and protected members without being a member.
- Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.

friend_function.cpp

```cpp
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

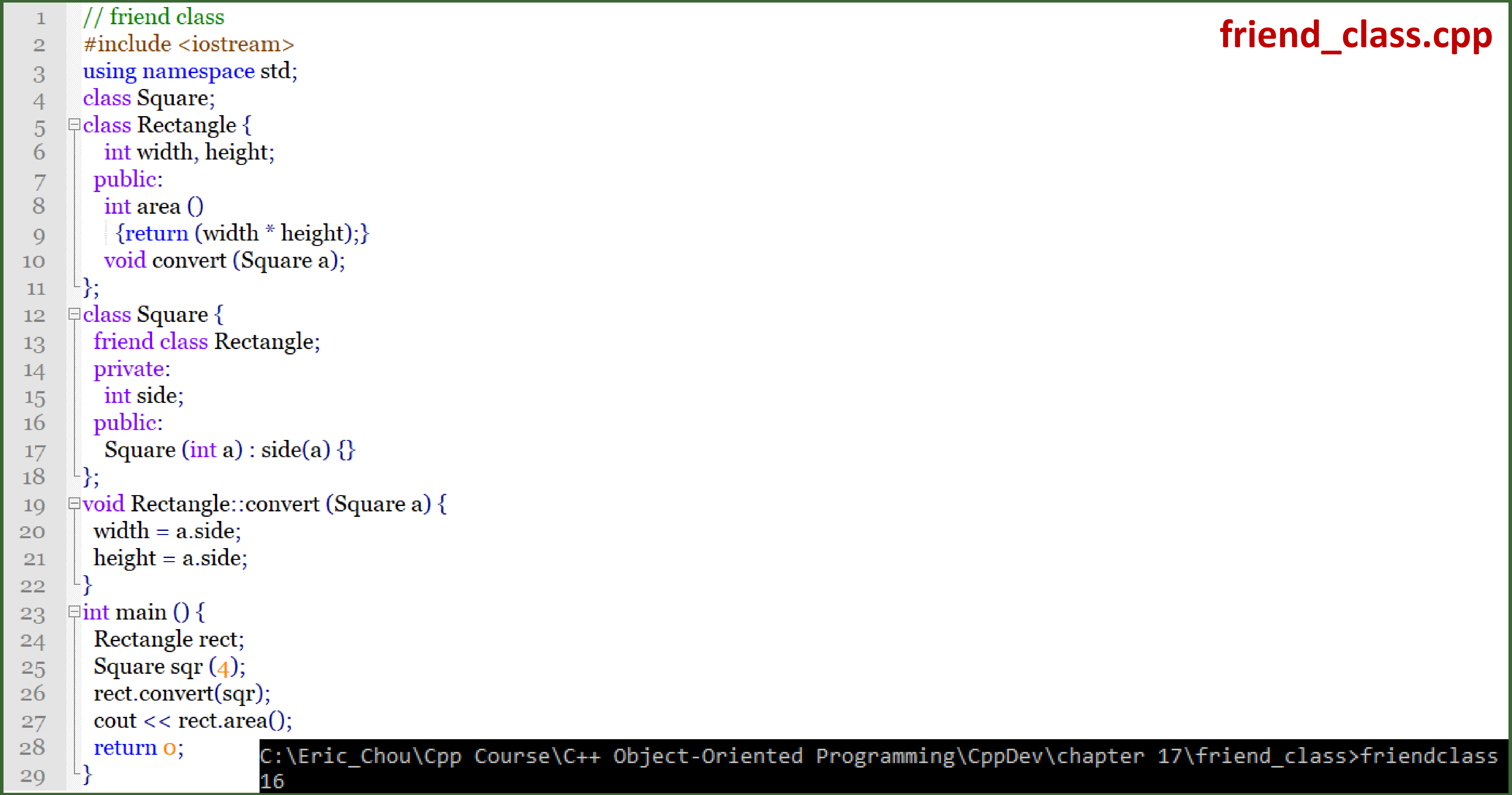
```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 17\friend_function>friendfunc
24
```

LECTURE 12

C++ Friend Classes

Friend Class

- Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class.



friend_class.cpp

```cpp
// friend class
#include <iostream>
using namespace std;
class Square;
class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};
class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};
void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}
int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 17\friend_class>friendclass
16
```

Demo Program: friend_class.cpp

- In this example, class **Rectangle** is a friend of class **Square** allowing Rectangle's member functions to access private and protected members of Square.
- More concretely, Rectangle accesses the member variable Square::side, which describes the side of the square.
- There is something else new in this example: at the beginning of the program, there is an empty declaration of class Square.
- This is necessary because class Rectangle uses Square (as a parameter in member convert), and Square uses Rectangle (declaring it a friend).

Demo Program: friend_class.cpp

- Friendships are never corresponded unless specified: In our example, Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle.
- Therefore, the member functions of Rectangle can access the protected and private members of Square but not the other way around. Of course, Square could also be declared friend of Rectangle, if needed, granting such an access.
- Another property of friendships is that they are not transitive: The friend of a friend is not considered a friend unless explicitly specified.