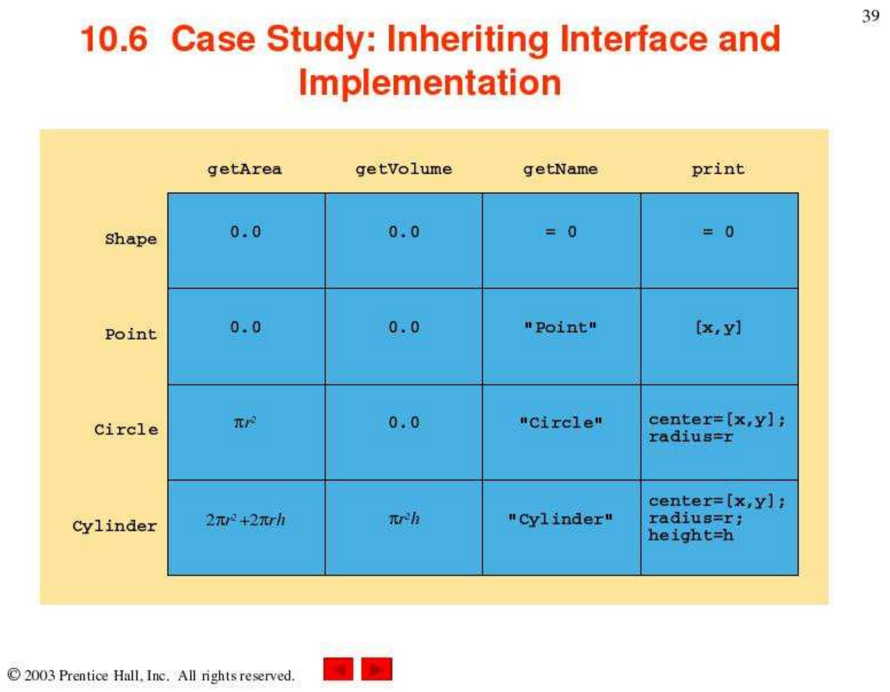
0.1 Case Study: Inheriting Interface and Implementation

Make abstract base class Shape

- Pure virtual functions (must be implemented)
  - **getName**, **print**
  - Default implementation does not make sense
- Virtual functions (may be redefined)
  - **getArea**, **getVolume**; initially return **0.0**
  - If not redefined, uses base class definition
- Derive classes **Point**, **Circle**, **Cylinder**



39

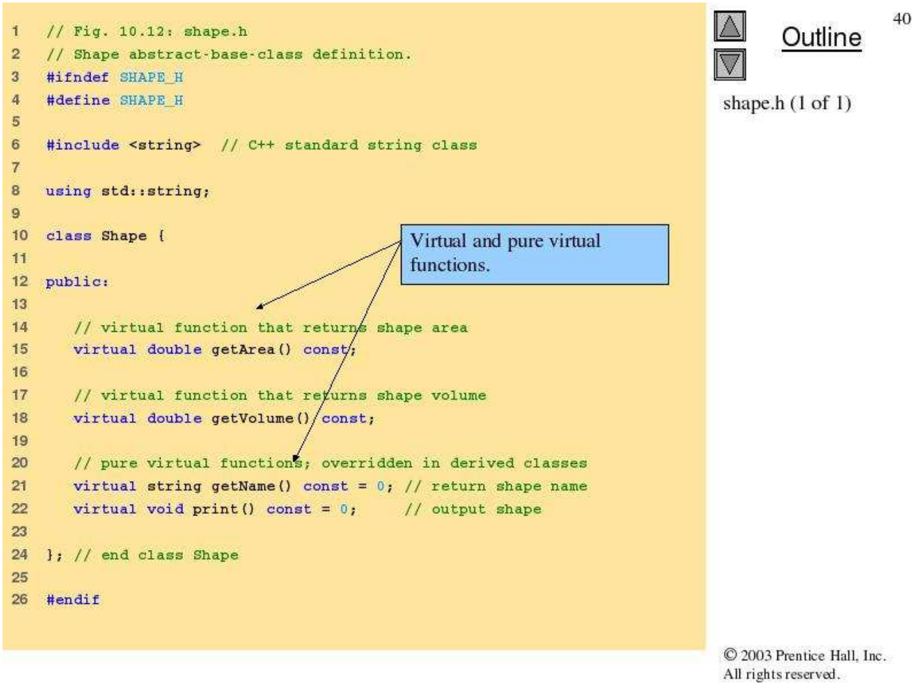
10.6 Case Study: Inheriting Interface and Implementation

	getArea	getVolume	getName	print
Shape	0.0	0.0	= 0	= 0
Point	0.0	0.0	"Point"	[x,y]

Circle	πr^2	0.0	"Circle"	center=[x,y];
				radius=r

Cylinder	$2\pi r^2+2\pi rh$	$\pi r^2 h$	"Cylinder"	center=[x,y];
				radius=r;
				height=h

Figure 1: Defining the polymorphic interface for the **Shape** hierarchy classes.



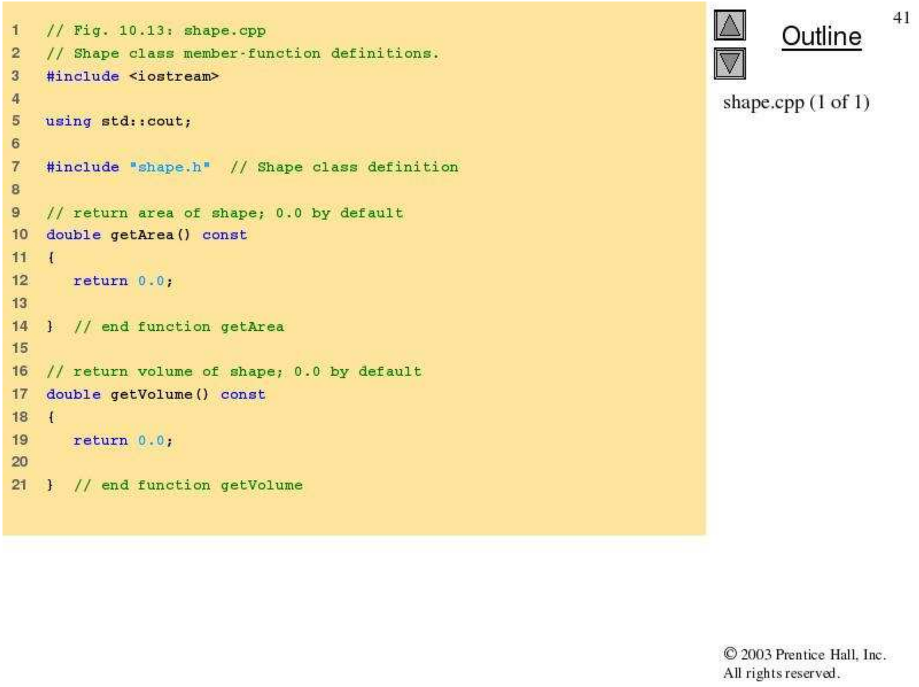
40

Outline

shape.h (1 of 1)

```
// Fig. 10.12: shape.h
// Shape abstract-base-class definition.
#ifndef SHAPE_H
#define SHAPE_H

#include <string>  // C++ standard string class

using std::string;

class Shape {

public:

   // virtual function that returns shape area
   virtual double getArea() const;

   // virtual function that returns shape volume
   virtual double getVolume() const;

   // pure virtual functions; overridden in derived classes
   virtual string getName() const = 0; // return shape name
   virtual void print() const = 0;     // output shape

}; // end class Shape

#endif
```

Virtual and pure virtual functions.



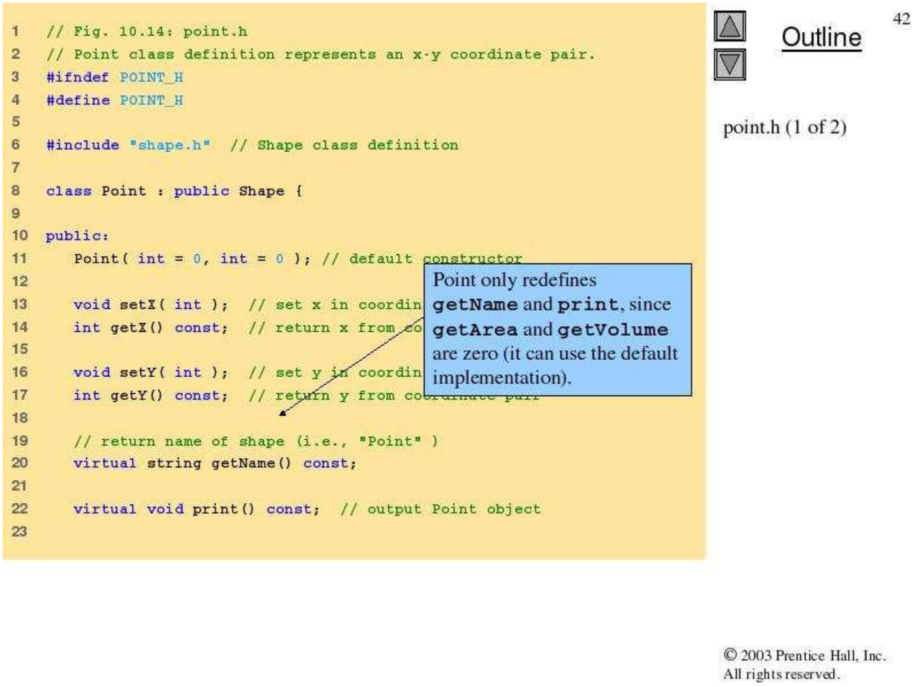
41

Outline

shape.cpp (1 of 1)

```
// Fig. 10.13: shape.cpp
// Shape class member-function definitions.
#include <iostream>

using std::cout;

#include "shape.h"  // Shape class definition

// return area of shape; 0.0 by default
double getArea() const
{
   return 0.0;

}  // end function getArea

// return volume of shape; 0.0 by default
double getVolume() const
{
   return 0.0;

}  // end function getVolume
```

Figure 2: Abstract base class **Shape** header file and Abstract base class **Shape**.



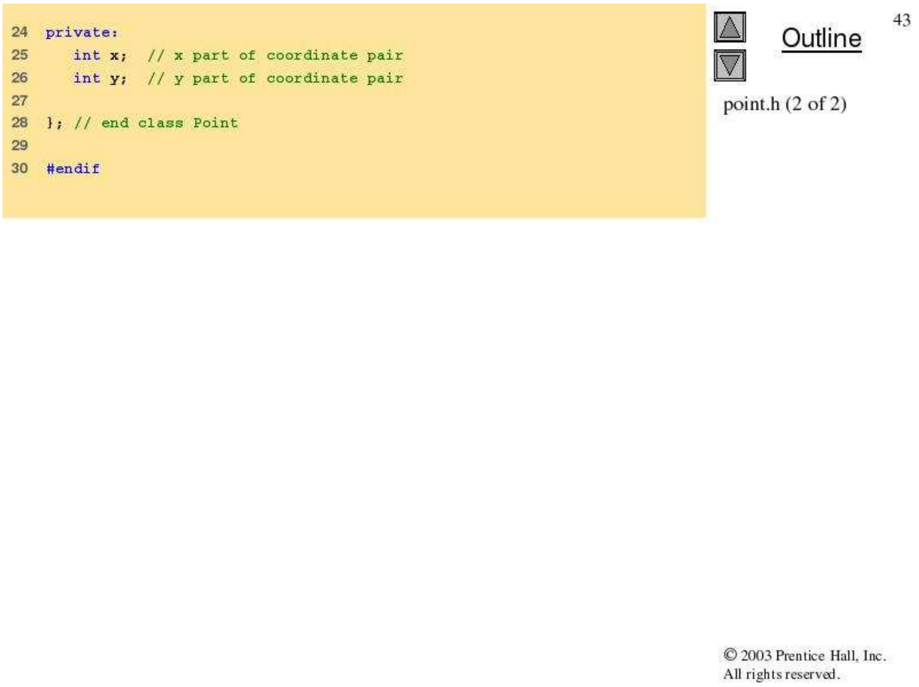
```
// Fig. 10.14: point.h
// Point class definition represents an x-y coordinate pair.
#ifndef POINT_H
#define POINT_H

#include "shape.h"  // Shape class definition

class Point : public Shape {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordin
   int getX() const;  // return x from co

   void setY( int );  // set y in coordin
   int getY() const;  // return y from coordinate pair

   // return name of shape (i.e., "Point" )
   virtual string getName() const;

   virtual void print() const;  // output Point object

```

Point only redefines **getName** and **print**, since **getArea** and **getVolume** are zero (it can use the default implementation).

42

Outline

point.h (1 of 2)



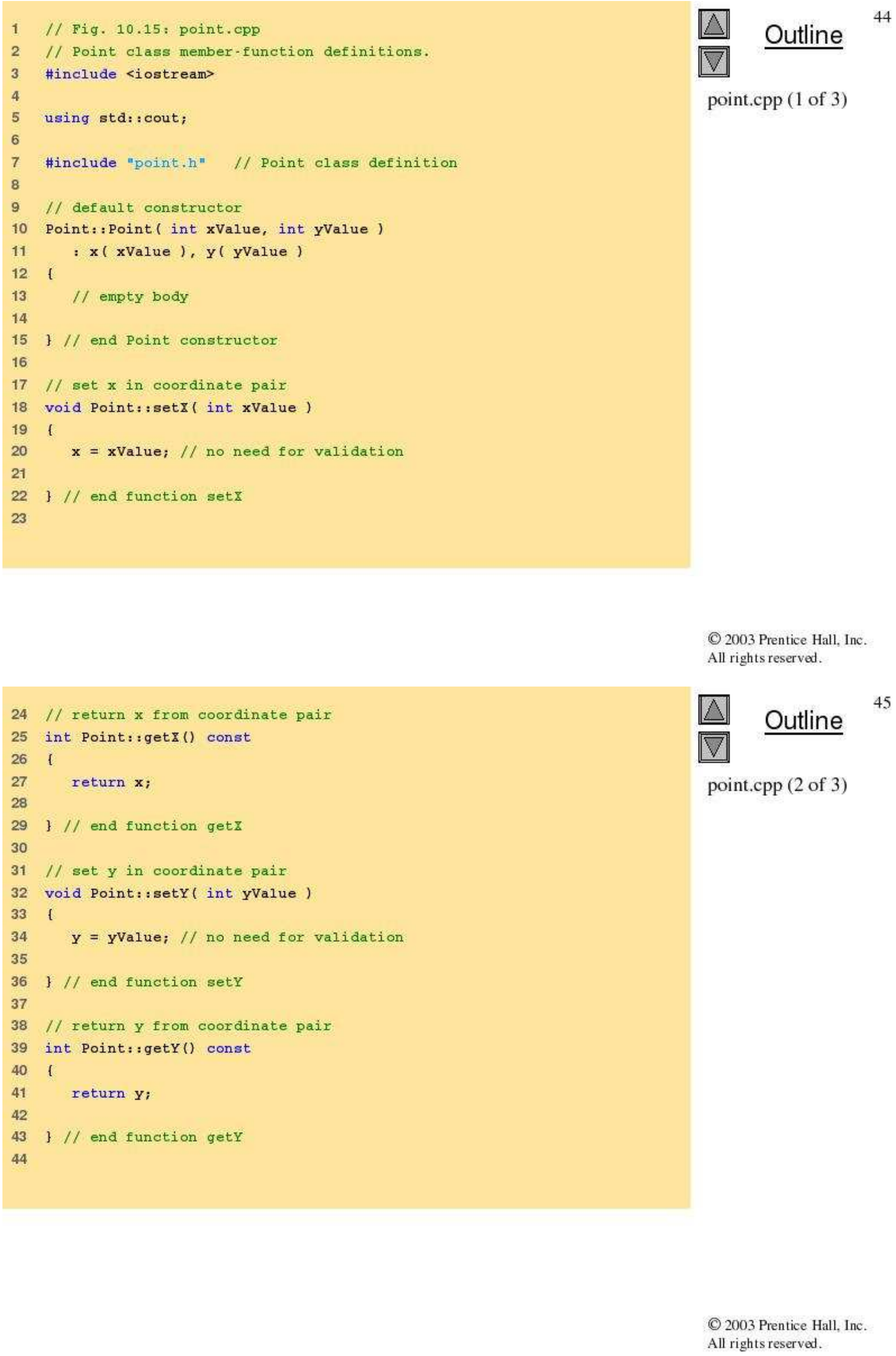
```
private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point

#endif
```

43

Outline

point.h (2 of 2)

Figure 3: **Point** class header file.



Outline

point.cpp (1 of 3)

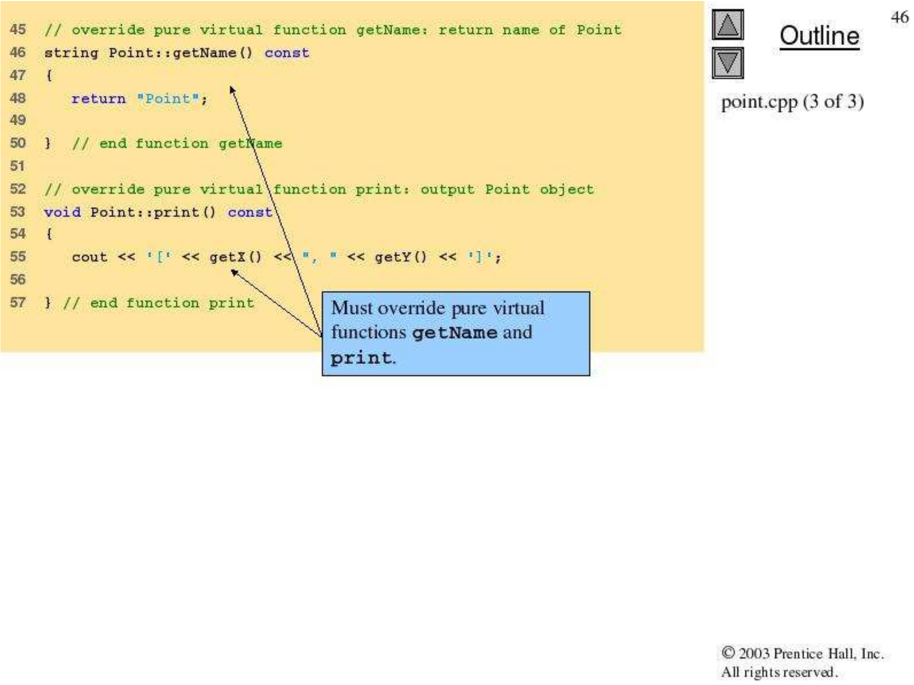
```cpp
// Fig. 10.15: point.cpp
// Point class member-function definitions.
#include <iostream>

using std::cout;

#include "point.h"   // Point class definition

// default constructor
Point::Point( int xValue, int yValue )
   : x( xValue ), y( yValue )
{
   // empty body

} // end Point constructor

// set x in coordinate pair
void Point::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

```

Outline

point.cpp (2 of 3)

```cpp
// return x from coordinate pair
int Point::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point::getY() const
{
   return y;

} // end function getY

```

Figure 4: **Point** class implementation file. (part 1 of 2)



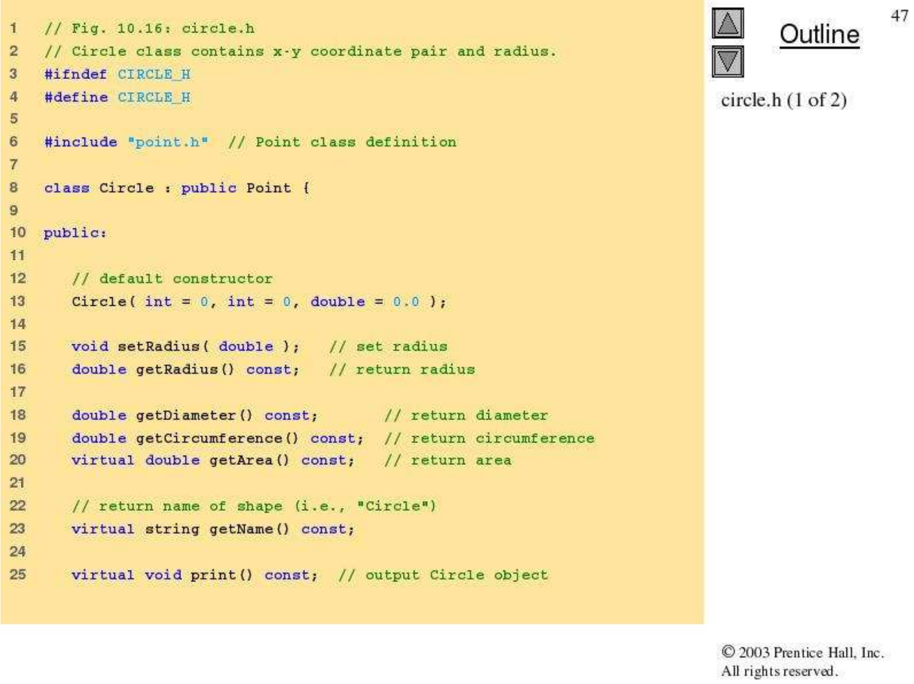
46

Outline

point.cpp (3 of 3)

```
// override pure virtual function getName: return name of Point
string Point::getName() const
{
   return "Point";

}  // end function getName

// override pure virtual function print: output Point object
void Point::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';

} // end function print
```

Must override pure virtual functions **getName** and **print**.



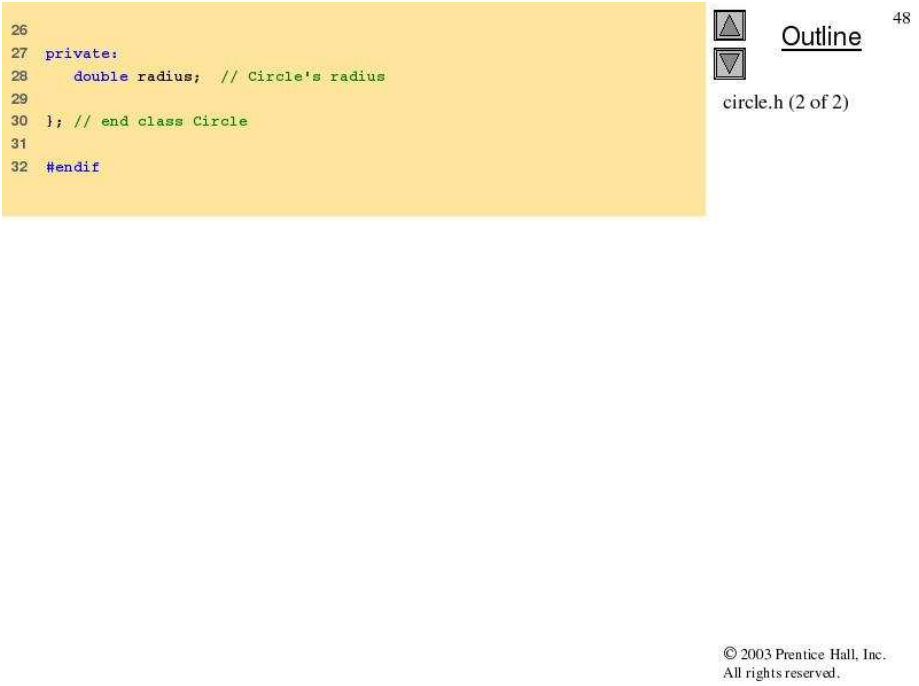
47

Outline

circle.h (1 of 2)

```
// Fig. 10.16: circle.h
// Circle class contains x-y coordinate pair and radius.
#ifndef CIRCLE_H
#define CIRCLE_H

#include "point.h"  // Point class definition

class Circle : public Point {

public:

   // default constructor
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   virtual double getArea() const;   // return area

   // return name of shape (i.e., "Circle")
   virtual string getName() const;

   virtual void print() const;  // output Circle object
```

Figure 5: **Point** class implementation file. (part 2 of 2)

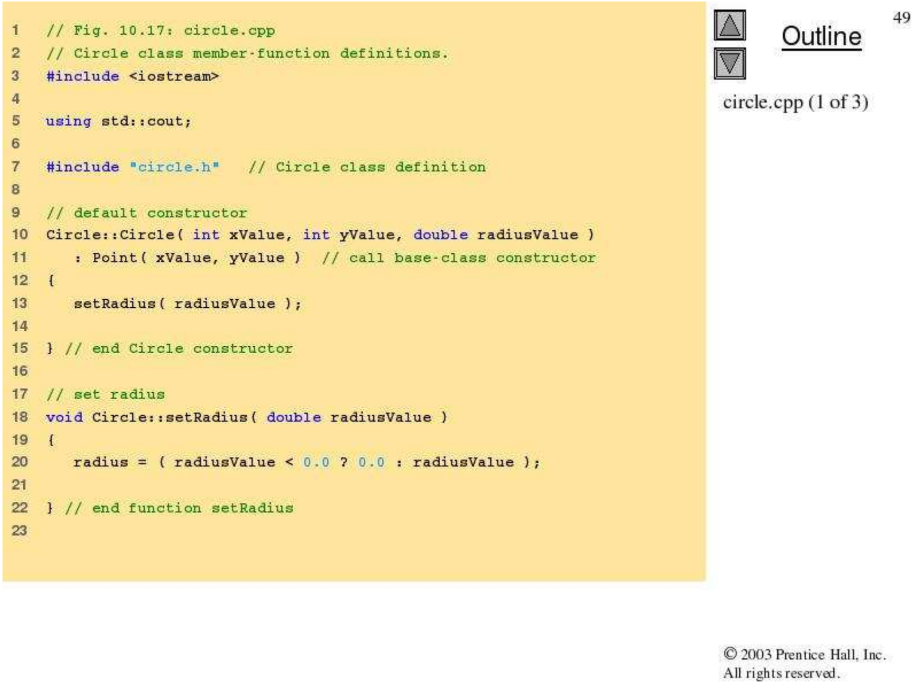


48

Outline

circle.h (2 of 2)

```cpp

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```



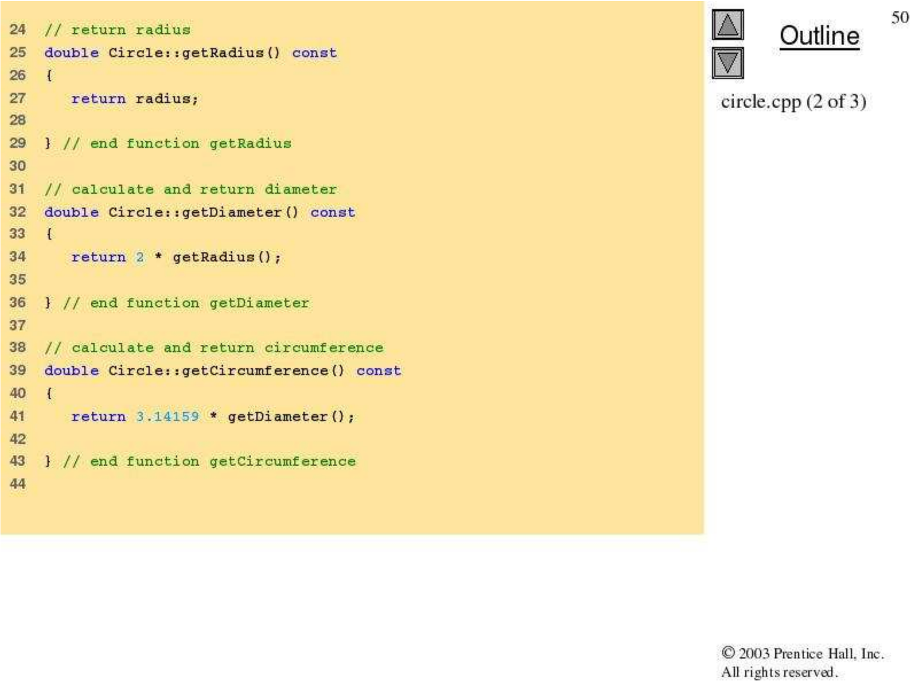
49

Outline

circle.cpp (1 of 3)

```cpp
// Fig. 10.17: circle.cpp
// Circle class member-function definitions.
#include <iostream>

using std::cout;

#include "circle.h"   // Circle class definition

// default constructor
Circle::Circle( int xValue, int yValue, double radiusValue )
   : Point( xValue, yValue )  // call base-class constructor
{
   setRadius( radiusValue );

} // end Circle constructor

// set radius
void Circle::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

```

Figure 6: **Circle** class header file and **Circle** class that inherits from class **Point**. (part 1 of 2)

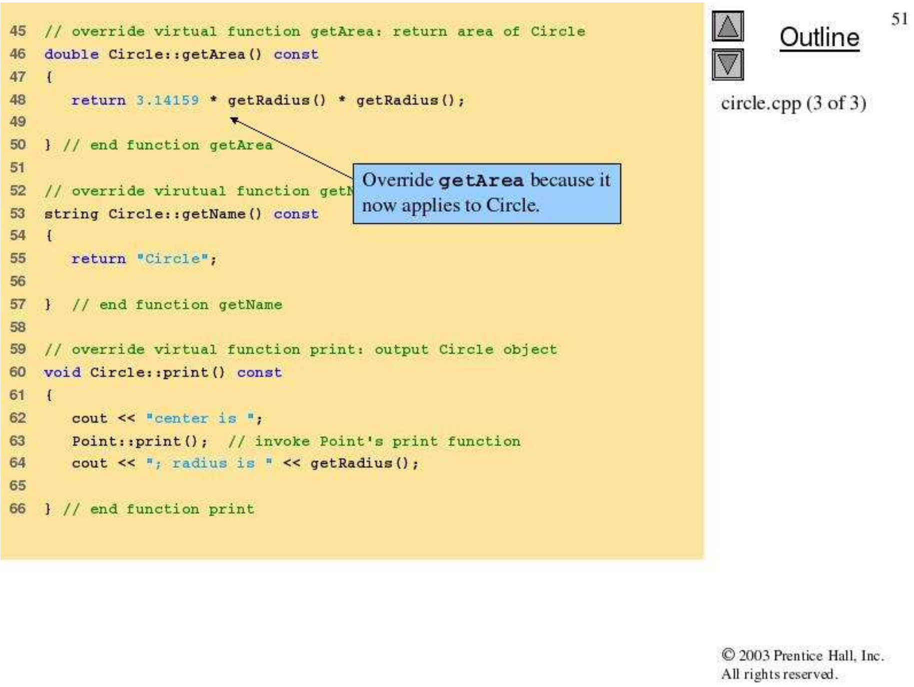


50

Outline

circle.cpp (2 of 3)

```
// return radius
double Circle::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle::getDiameter() const
{
   return 2 * getRadius();

} // end function getDiameter

// calculate and return circumference
double Circle::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

```



51

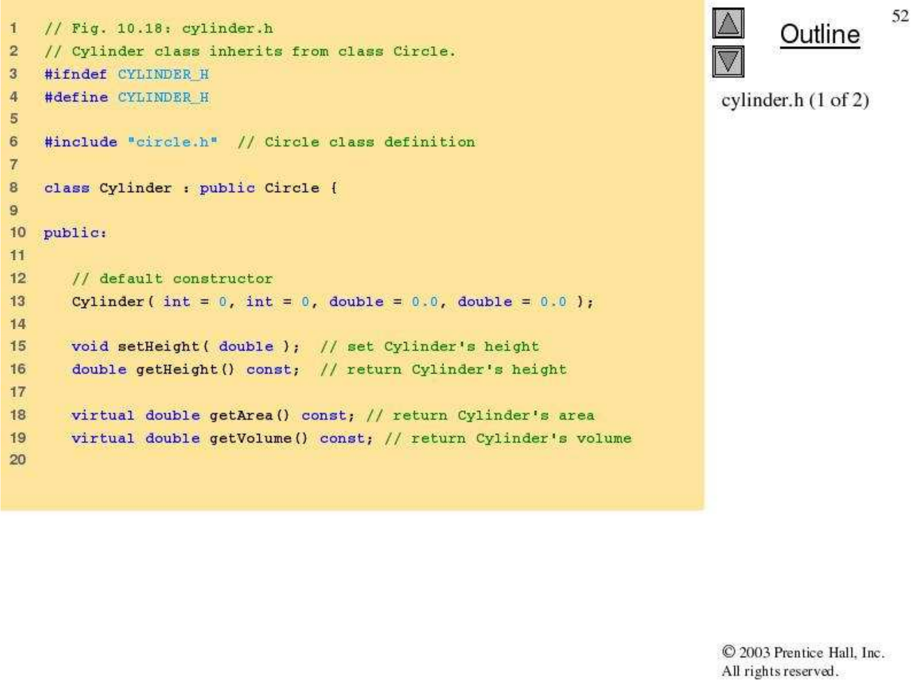
Outline

circle.cpp (3 of 3)

```
// override virtual function getArea: return area of Circle
double Circle::getArea() const
{
   return 3.14159 * getRadius() * getRadius();

} // end function getArea

// override virutual function getN
string Circle::getName() const
{
   return "Circle";

}  // end function getName

// override virtual function print: output Circle object
void Circle::print() const
{
   cout << "center is ";
   Point::print();  // invoke Point's print function
   cout << "; radius is " << getRadius();

} // end function print
```

Override **getArea** because it now applies to Circle.

Figure 7: **Circle** class that inherits from class **Point**. (part 2 of 2)

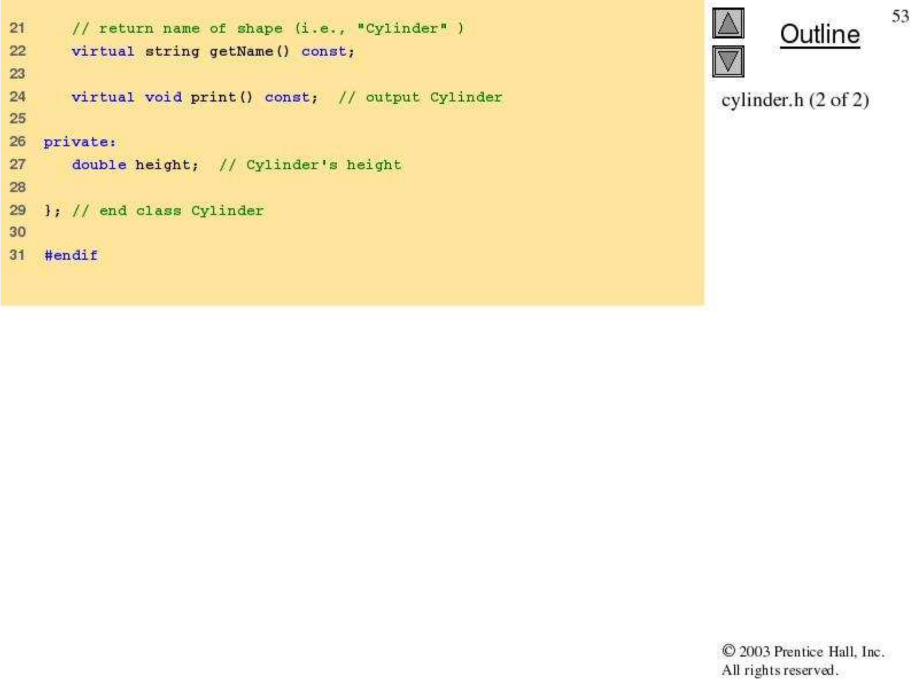


52

Outline

cylinder.h (1 of 2)

```
// Fig. 10.18: cylinder.h
// Cylinder class inherits from class Circle.
#ifndef CYLINDER_H
#define CYLINDER_H

#include "circle.h"  // Circle class definition

class Cylinder : public Circle {

public:

   // default constructor
   Cylinder( int = 0, int = 0, double = 0.0, double = 0.0 );

   void setHeight( double );  // set Cylinder's height
   double getHeight() const;  // return Cylinder's height

   virtual double getArea() const; // return Cylinder's area
   virtual double getVolume() const; // return Cylinder's volume

```



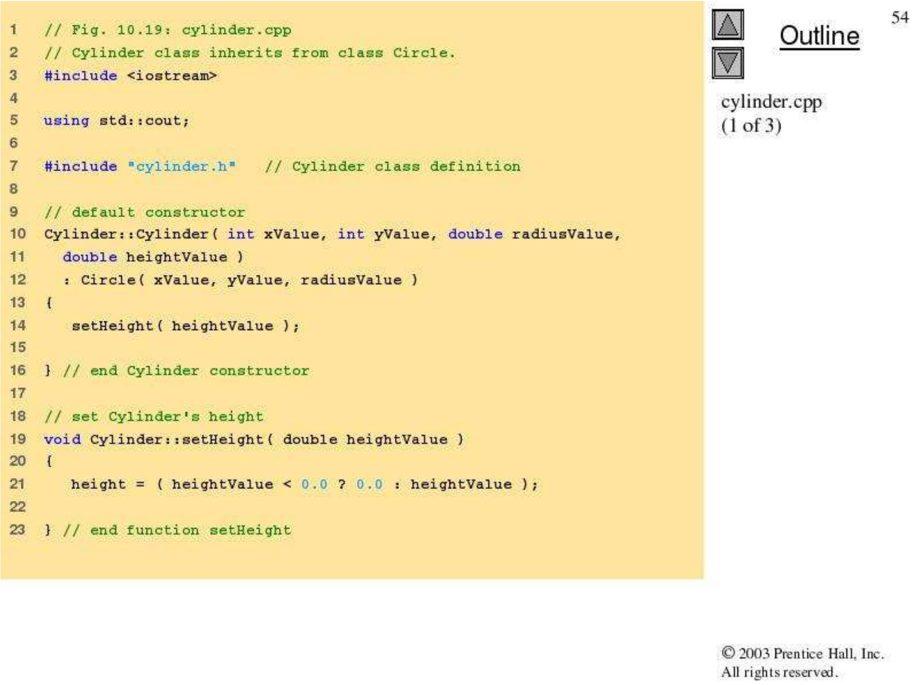
53

Outline

cylinder.h (2 of 2)

```
   // return name of shape (i.e., "Cylinder" )
   virtual string getName() const;

   virtual void print() const;  // output Cylinder

private:
   double height;  // Cylinder's height

}; // end class Cylinder

#endif
```

Figure 8: **Cylinder** class header file.

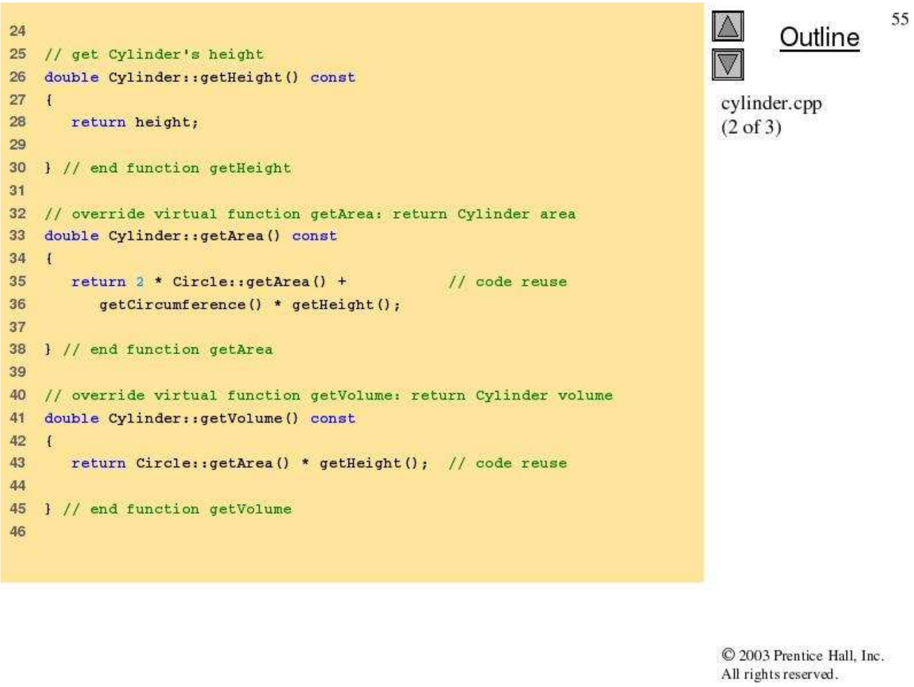


54

Outline

cylinder.cpp
(1 of 3)

```cpp
// Fig. 10.19: cylinder.cpp
// Cylinder class inherits from class Circle.
#include <iostream>

using std::cout;

#include "cylinder.h"   // Cylinder class definition

// default constructor
Cylinder::Cylinder( int xValue, int yValue, double radiusValue,
  double heightValue )
  : Circle( xValue, yValue, radiusValue )
{
   setHeight( heightValue );

} // end Cylinder constructor

// set Cylinder's height
void Cylinder::setHeight( double heightValue )
{
   height = ( heightValue < 0.0 ? 0.0 : heightValue );

} // end function setHeight
```



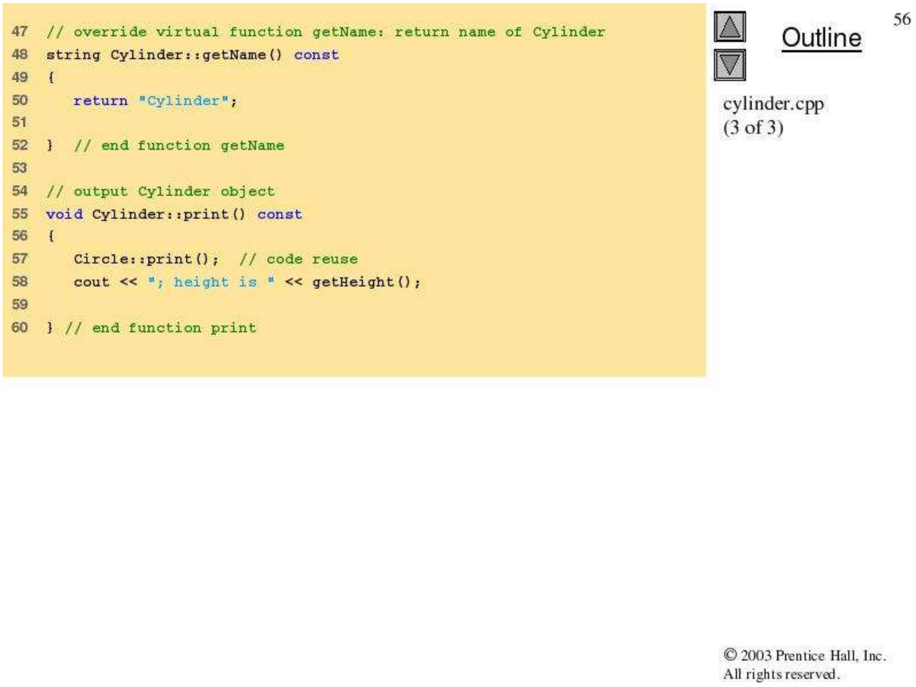
55

Outline

cylinder.cpp
(2 of 3)

```cpp

// get Cylinder's height
double Cylinder::getHeight() const
{
   return height;

} // end function getHeight

// override virtual function getArea: return Cylinder area
double Cylinder::getArea() const
{
   return 2 * Circle::getArea() +           // code reuse
      getCircumference() * getHeight();

} // end function getArea

// override virtual function getVolume: return Cylinder volume
double Cylinder::getVolume() const
{
   return Circle::getArea() * getHeight();  // code reuse

} // end function getVolume

```

Figure 9: **Cylinder** class implementation file. (part 1 of 2)

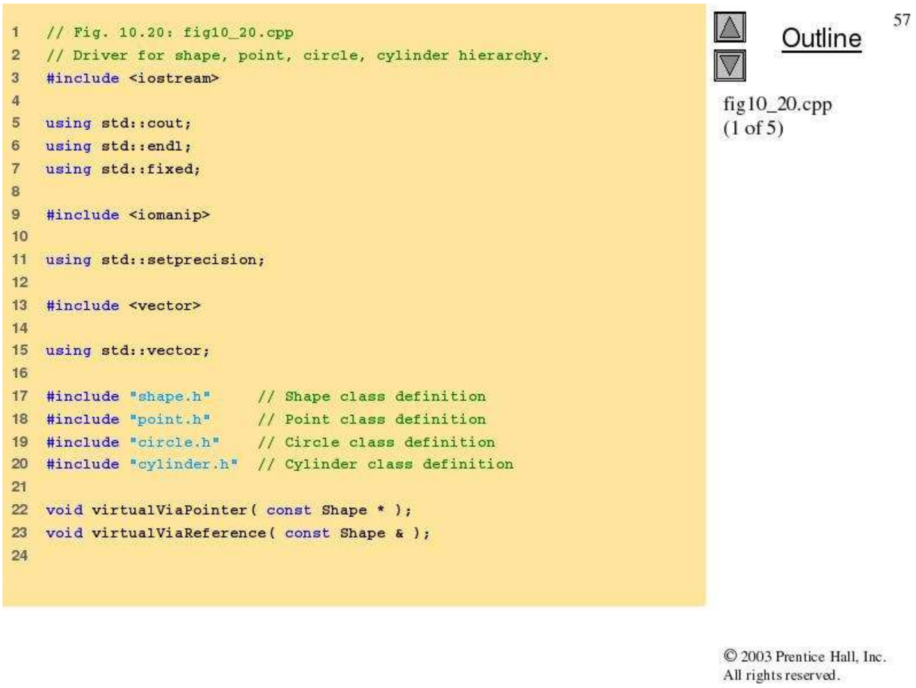


56

Outline

cylinder.cpp
(3 of 3)

```cpp
// override virtual function getName: return name of Cylinder
string Cylinder::getName() const
{
   return "Cylinder";

}  // end function getName

// output Cylinder object
void Cylinder::print() const
{
   Circle::print();  // code reuse
   cout << "; height is " << getHeight();

} // end function print
```



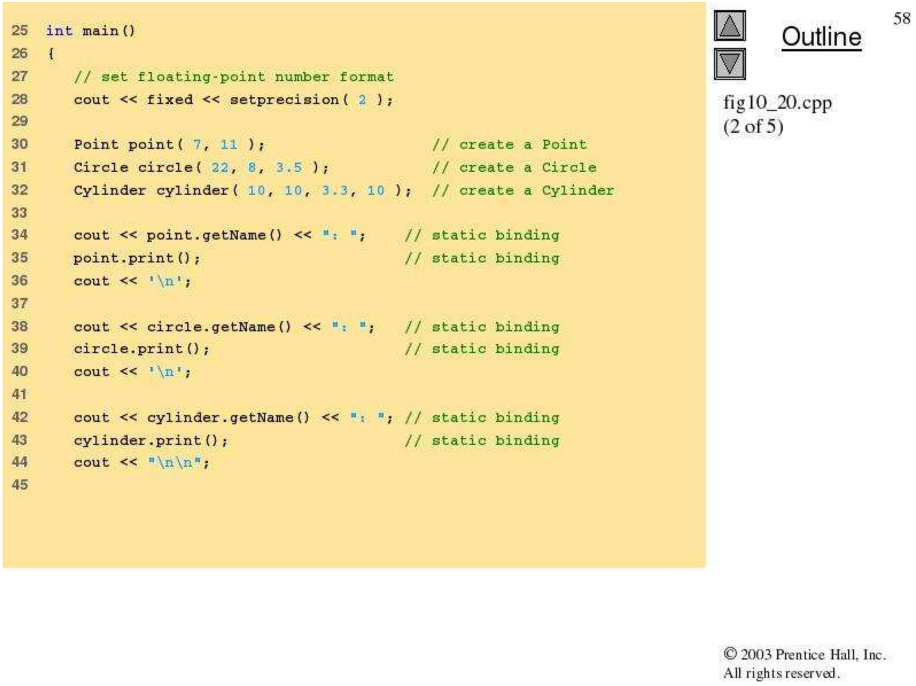
57

Outline

fig10_20.cpp
(1 of 5)

```cpp
// Fig. 10.20: fig10_20.cpp
// Driver for shape, point, circle, cylinder hierarchy.
#include <iostream>

using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>

using std::setprecision;

#include <vector>

using std::vector;

#include "shape.h"     // Shape class definition
#include "point.h"     // Point class definition
#include "circle.h"    // Circle class definition
#include "cylinder.h"  // Cylinder class definition

void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );

```

Figure 10: **Cylinder** class implementation file. (part 2 of 2)

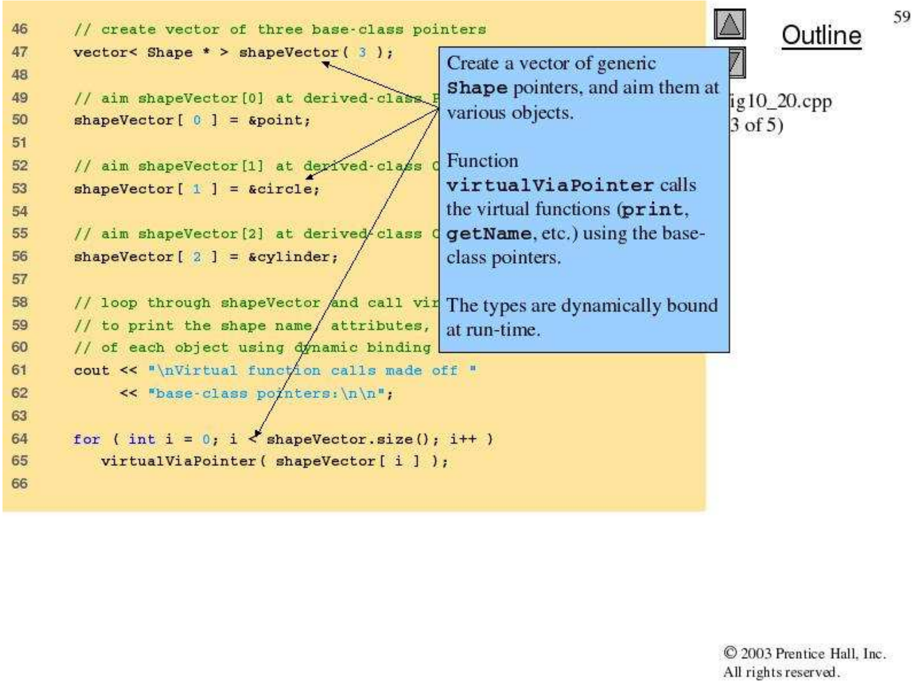


58

Outline

fig10_20.cpp
(2 of 5)

```
int main()
{
   // set floating-point number format
   cout << fixed << setprecision( 2 );

   Point point( 7, 11 );                  // create a Point
   Circle circle( 22, 8, 3.5 );           // create a Circle
   Cylinder cylinder( 10, 10, 3.3, 10 );  // create a Cylinder

   cout << point.getName() << ": ";    // static binding
   point.print();                      // static binding
   cout << '\n';

   cout << circle.getName() << ": ";   // static binding
   circle.print();                     // static binding
   cout << '\n';

   cout << cylinder.getName() << ": "; // static binding
   cylinder.print();                   // static binding
   cout << "\n\n";

```



59

Outline

ig10_20.cpp
3 of 5)

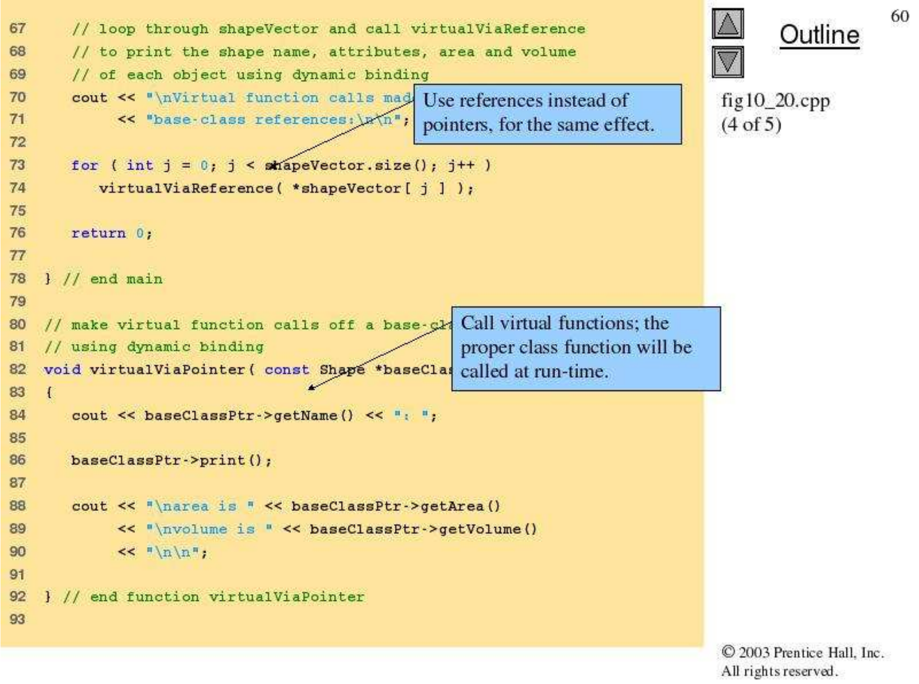
```
   // create vector of three base-class pointers
   vector< Shape * > shapeVector( 3 );

   // aim shapeVector[0] at derived-class P
   shapeVector[ 0 ] = &point;

   // aim shapeVector[1] at derived-class C
   shapeVector[ 1 ] = &circle;

   // aim shapeVector[2] at derived-class C
   shapeVector[ 2 ] = &cylinder;

   // loop through shapeVector and call vir
   // to print the shape name, attributes,
   // of each object using dynamic binding
   cout << "\nVirtual function calls made off "
        << "base-class pointers:\n\n";

   for ( int i = 0; i < shapeVector.size(); i++ )
      virtualViaPointer( shapeVector[ i ] );

```

Create a vector of generic **Shape** pointers, and aim them at various objects.

Function **virtualViaPointer** calls the virtual functions (**print**, **getName**, etc.) using the base-class pointers.

The types are dynamically bound at run-time.

Figure 11: Demonstarting polymorphism via a hierarchy headed by an abstract base class. (part 1 of 3)



60

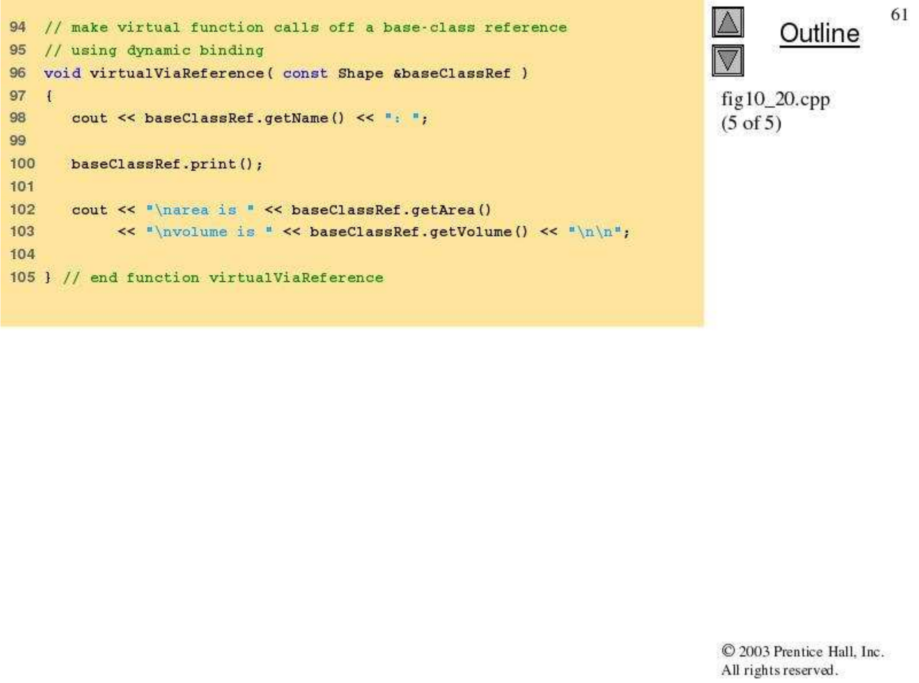
Outline

fig10_20.cpp
(4 of 5)

```cpp
   // loop through shapeVector and call virtualViaReference
   // to print the shape name, attributes, area and volume
   // of each object using dynamic binding
   cout << "\nVirtual function calls mad
        << "base-class references:\n\n";

   for ( int j = 0; j < shapeVector.size(); j++ )
      virtualViaReference( *shapeVector[ j ] );

   return 0;

} // end main

// make virtual function calls off a base-cl
// using dynamic binding
void virtualViaPointer( const Shape *baseCla
{
   cout << baseClassPtr->getName() << ": ";

   baseClassPtr->print();

   cout << "\narea is " << baseClassPtr->getArea()
        << "\nvolume is " << baseClassPtr->getVolume()
        << "\n\n";

} // end function virtualViaPointer

```

Use references instead of pointers, for the same effect.

Call virtual functions; the proper class function will be called at run-time.



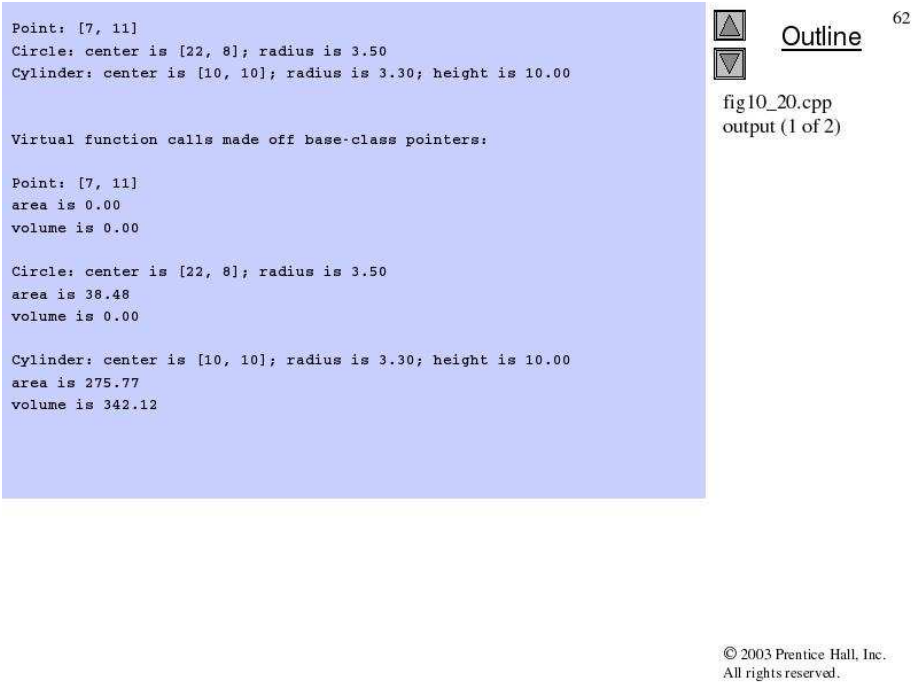
61

Outline

fig10_20.cpp
(5 of 5)

```cpp
// make virtual function calls off a base-class reference
// using dynamic binding
void virtualViaReference( const Shape &baseClassRef )
{
   cout << baseClassRef.getName() << ": ";

   baseClassRef.print();

   cout << "\narea is " << baseClassRef.getArea()
        << "\nvolume is " << baseClassRef.getVolume() << "\n\n";

} // end function virtualViaReference
```

Figure 12: Demonstarting polymorphism via a hierarchy headed by an abstract base class. (part 2 of 3)

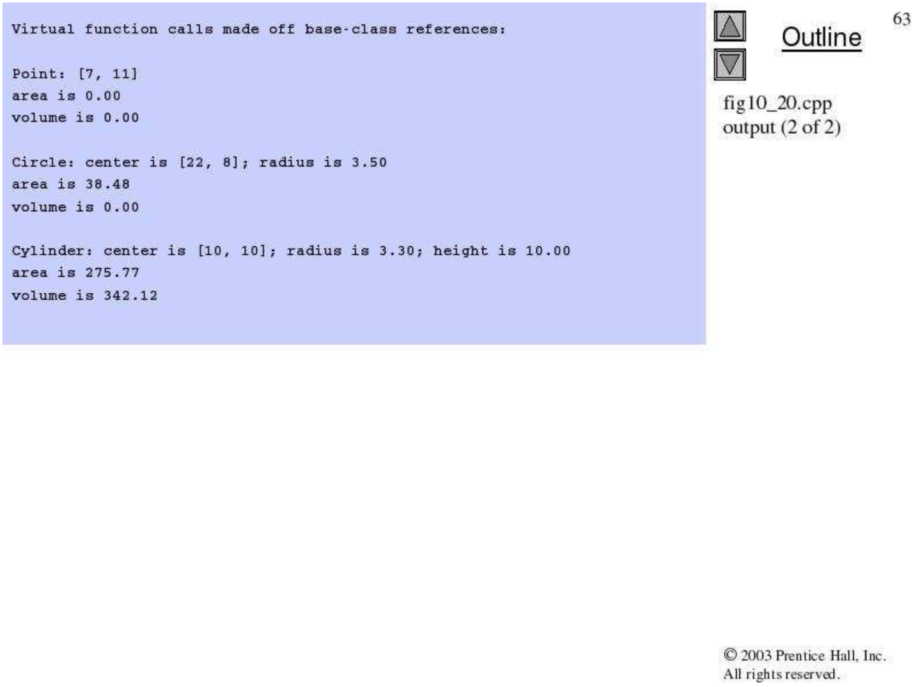


62

Outline

fig10_20.cpp
output (1 of 2)

```
Point: [7, 11]
Circle: center is [22, 8]; radius is 3.50
Cylinder: center is [10, 10]; radius is 3.30; height is 10.00


Virtual function calls made off base-class pointers:

Point: [7, 11]
area is 0.00
volume is 0.00

Circle: center is [22, 8]; radius is 3.50
area is 38.48
volume is 0.00

Cylinder: center is [10, 10]; radius is 3.30; height is 10.00
area is 275.77
volume is 342.12
```



63

Outline

fig10_20.cpp
output (2 of 2)

```
Virtual function calls made off base-class references:

Point: [7, 11]
area is 0.00
volume is 0.00

Circle: center is [22, 8]; radius is 3.50
area is 38.48
volume is 0.00

Cylinder: center is [10, 10]; radius is 3.30; height is 10.00
area is 275.77
volume is 342.12
```

Figure 13: Demonstarting polymorphism via a hierarchy headed by an abstract base class. (part 3 of 3)

0.2 Polymorphism, Virtual Functions and Dynamic Binding ”Under the Hood”

- Polymorphism has overhead
  - Not used in STL (Standard Template Library) to optimize performance
- virtual function table (vtable)
  - Every class with a **virtual** function has a vtable
  - For every **virtual** function, vtable has pointer to the proper function
  - If derived class has same function as base class; function pointer aims at base-class function
  - Detailed explanation in Fig. 10.21 (in book) (will not be covered)

0.3 Virtual Destructors

- Base class pointer to derived object; if destroyed using **delete**, behavior unspecified
- Simple fix
  - Declare base-class destructor virtual; makes derived-class destructors virtual
  - Now, when **delete** used appropriate destructor called
- When derived-class object destroyed
  - Derived-class destructor executes first
  - Base-class destructor executes afterwards
- Constructors cannot be virtual

0.4 Case Study: Payroll System Using Polymorphism

- Base class Employee
  - Pure virtual function **earnings** (returns pay)
    - ∗ Pure virtual because need to know employee type
    - ∗ Cannot calculate for generic employee

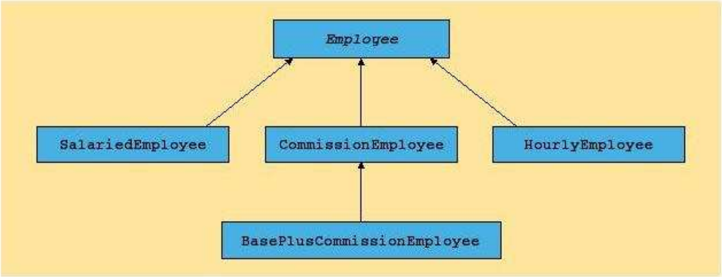
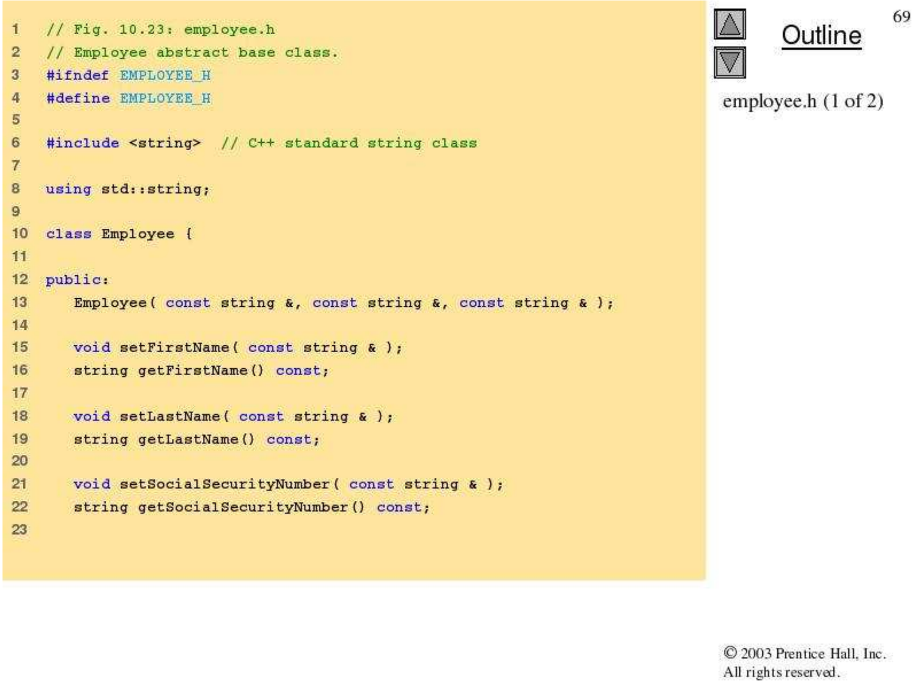


Figure 14: Class hierarchy for the polymorphic employee-payroll application.

  - Other classes derive from **Employee**

- Downcasting
  - **dynamic_cast** operator
    - Determine object's type at runtime
    - Returns 0 if not of proper type (cannot be cast)
    - **NewClass *ptr = dynamic_cast ¡ NewClass *¿ objectPtr;**
- Keyword typeid
  - Header **¡typeinfo¿**
  - Usage: **typeid(object)**
    - Returns **type_info** object
    - Has information about type of operand, including name
    - **typeid(object).name()**

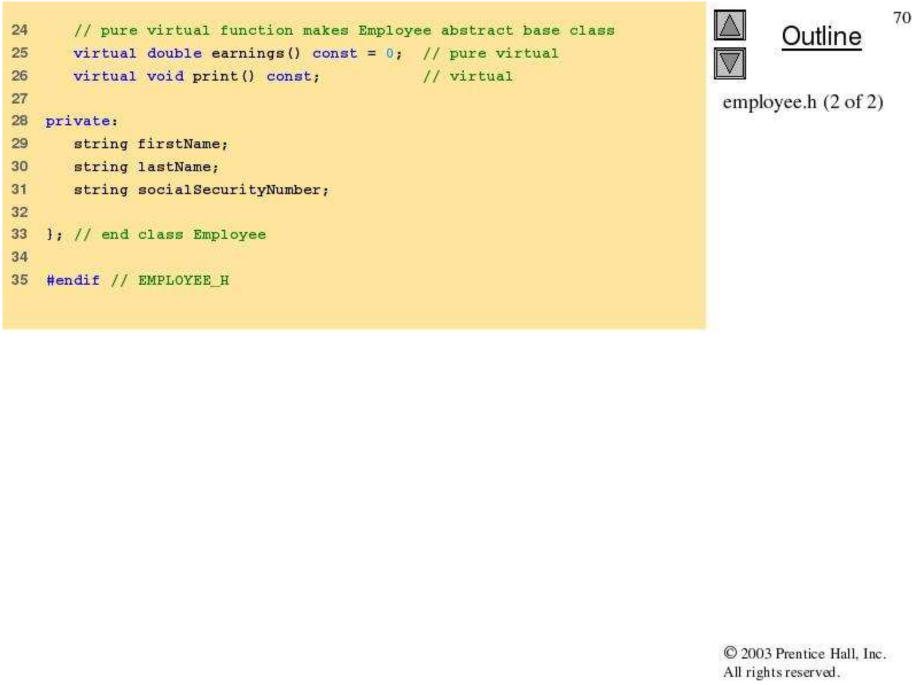


69

Outline

employee.h (1 of 2)

```cpp
// Fig. 10.23: employee.h
// Employee abstract base class.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>  // C++ standard string class

using std::string;

class Employee {

public:
   Employee( const string &, const string &, const string & );

   void setFirstName( const string & );
   string getFirstName() const;

   void setLastName( const string & );
   string getLastName() const;

   void setSocialSecurityNumber( const string & );
   string getSocialSecurityNumber() const;

```



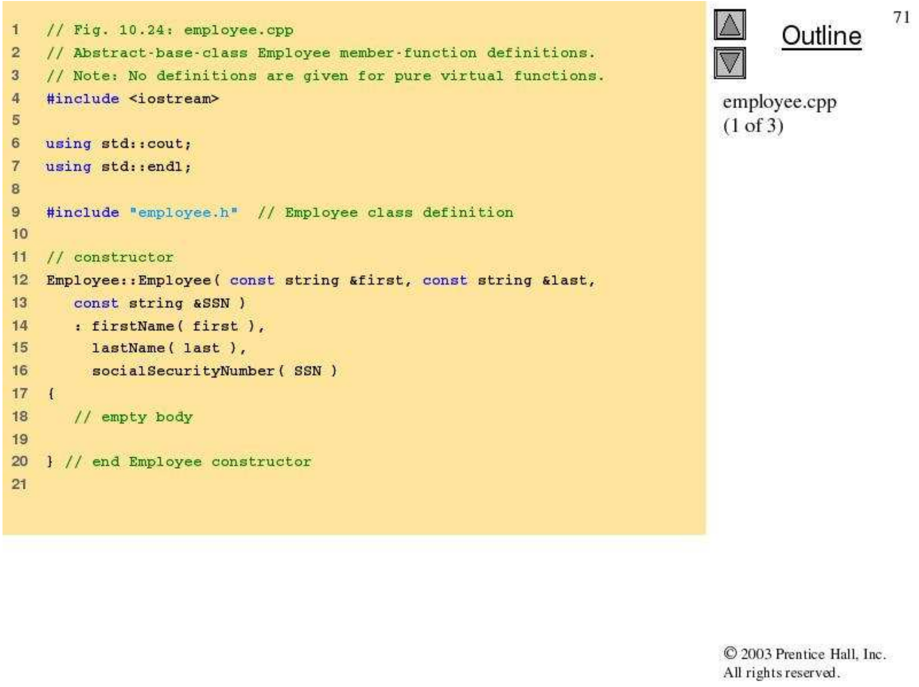
70

Outline

employee.h (2 of 2)

```cpp
   // pure virtual function makes Employee abstract base class
   virtual double earnings() const = 0;  // pure virtual
   virtual void print() const;           // virtual

private:
   string firstName;
   string lastName;
   string socialSecurityNumber;

}; // end class Employee

#endif // EMPLOYEE_H
```

Figure 15: **Employee** class header file.

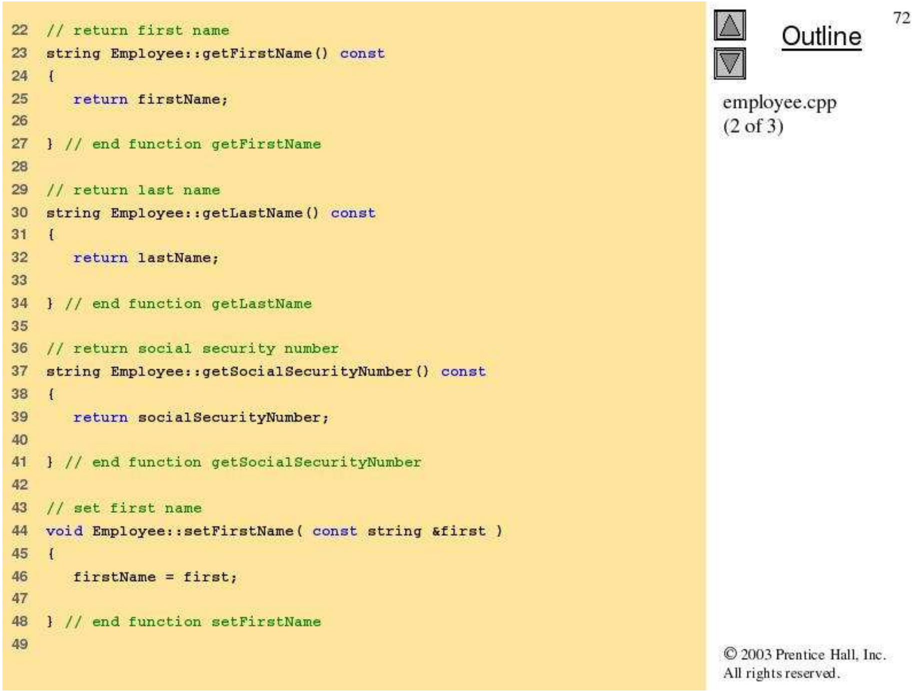


71

Outline

employee.cpp
(1 of 3)

```cpp
// Fig. 10.24: employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <iostream>

using std::cout;
using std::endl;

#include "employee.h"  // Employee class definition

// constructor
Employee::Employee( const string &first, const string &last,
   const string &SSN )
   : firstName( first ),
     lastName( last ),
     socialSecurityNumber( SSN )
{
   // empty body

} // end Employee constructor

```



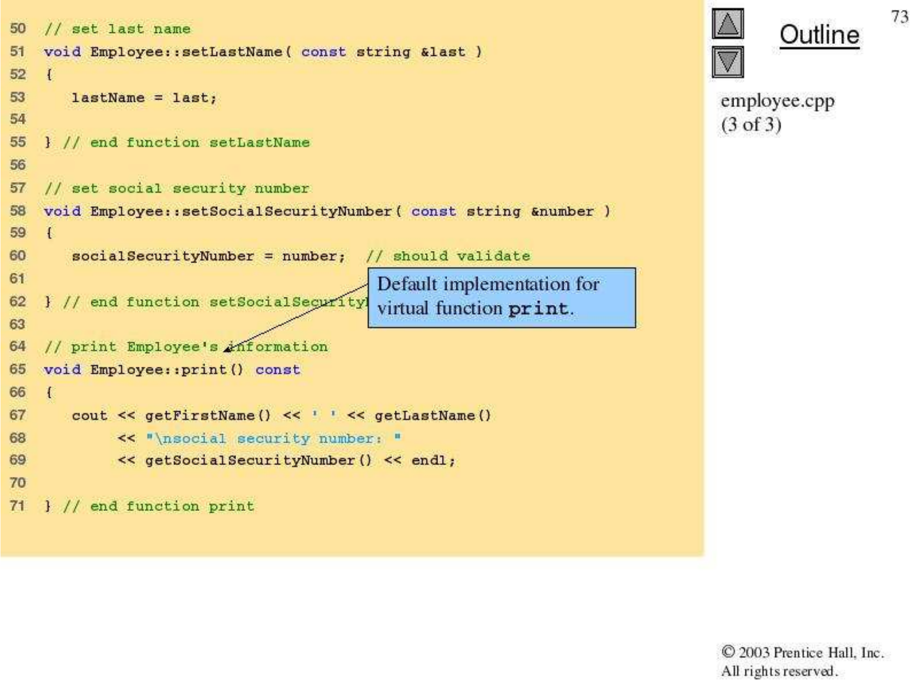
72

Outline

employee.cpp
(2 of 3)

```cpp
// return first name
string Employee::getFirstName() const
{
   return firstName;

} // end function getFirstName

// return last name
string Employee::getLastName() const
{
   return lastName;

} // end function getLastName

// return social security number
string Employee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;

} // end function getSocialSecurityNumber

// set first name
void Employee::setFirstName( const string &first )
{
   firstName = first;

} // end function setFirstName

```

Figure 16: **Employee** class implementation file. (part 1 of 2)



73

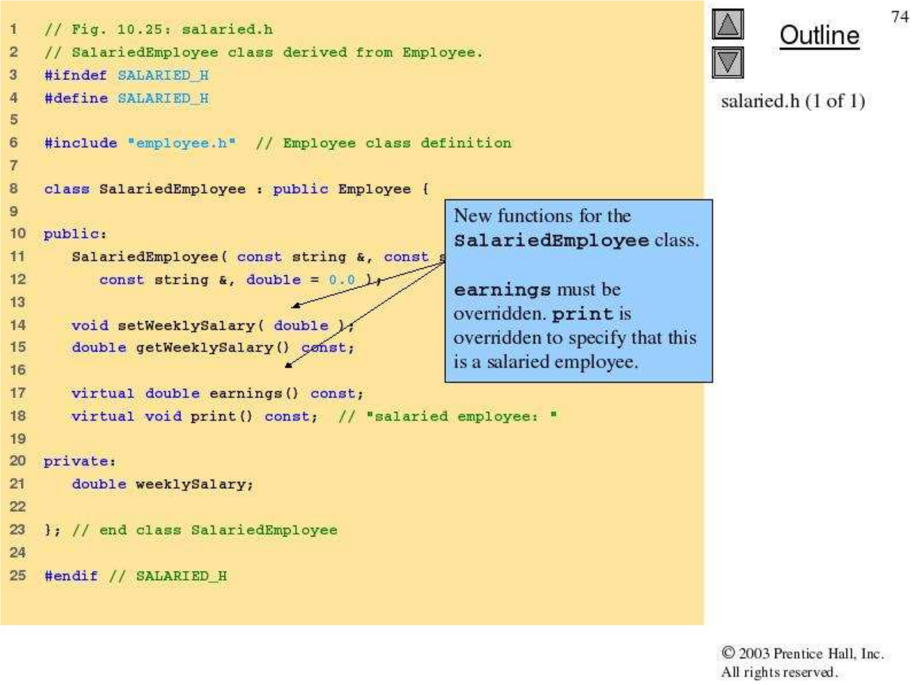
Outline

employee.cpp
(3 of 3)

```
// set last name
void Employee::setLastName( const string &last )
{
   lastName = last;

} // end function setLastName

// set social security number
void Employee::setSocialSecurityNumber( const string &number )
{
   socialSecurityNumber = number;  // should validate

} // end function setSocialSecurity

// print Employee's information
void Employee::print() const
{
   cout << getFirstName() << ' ' << getLastName()
        << "\nsocial security number: "
        << getSocialSecurityNumber() << endl;

} // end function print
```

Default implementation for virtual function **print**.

© 2003 Prentice Hall, Inc.
All rights reserved.



74

Outline

salaried.h (1 of 1)

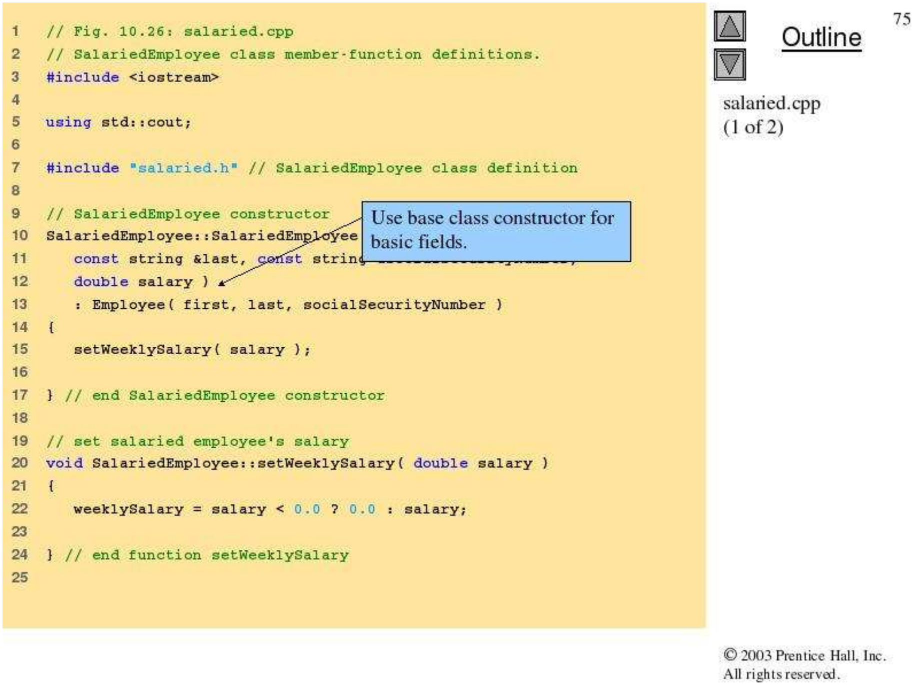
```
// Fig. 10.25: salaried.h
// SalariedEmployee class derived from Employee.
#ifndef SALARIED_H
#define SALARIED_H

#include "employee.h"  // Employee class definition

class SalariedEmployee : public Employee {

public:
   SalariedEmployee( const string &, const s
      const string &, double = 0.0 );

   void setWeeklySalary( double );
   double getWeeklySalary() const;

   virtual double earnings() const;
   virtual void print() const;  // "salaried employee: "

private:
   double weeklySalary;

}; // end class SalariedEmployee

#endif // SALARIED_H
```

New functions for the **SalariedEmployee** class.

earnings must be overridden. **print** is overridden to specify that this is a salaried employee.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 17: **Employee** class implementation file (part 2 of 2) and **SalariedEmployee** class header file.



75

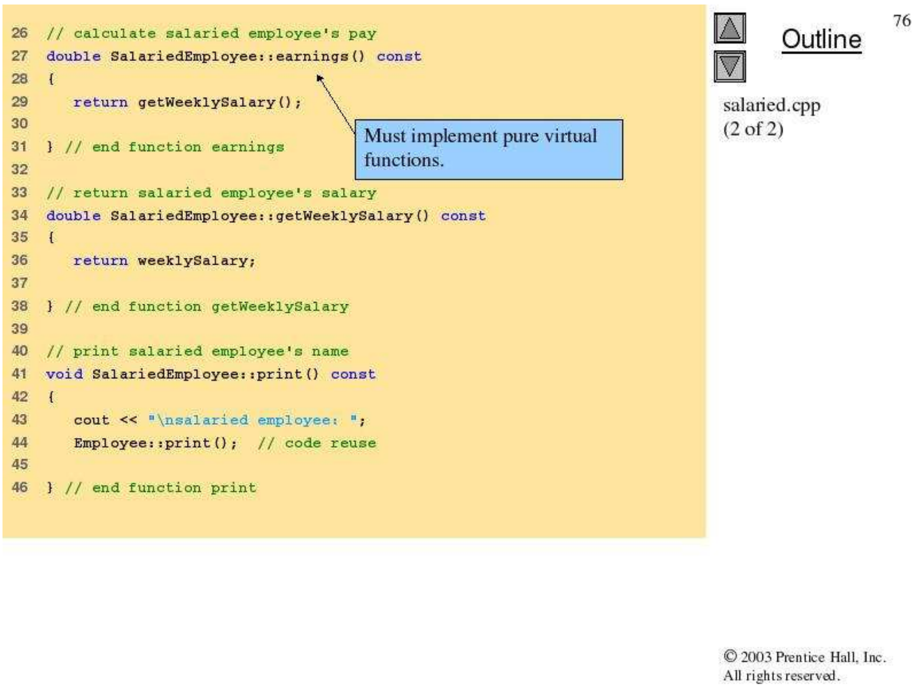
Outline

salaried.cpp
(1 of 2)

```
// Fig. 10.26: salaried.cpp
// SalariedEmployee class member-function definitions.
#include <iostream>

using std::cout;

#include "salaried.h" // SalariedEmployee class definition

// SalariedEmployee constructor
SalariedEmployee::SalariedEmployee
   const string &last, const string
   double salary )
   : Employee( first, last, socialSecurityNumber )
{
   setWeeklySalary( salary );

} // end SalariedEmployee constructor

// set salaried employee's salary
void SalariedEmployee::setWeeklySalary( double salary )
{
   weeklySalary = salary < 0.0 ? 0.0 : salary;

} // end function setWeeklySalary

```

Use base class constructor for basic fields.

© 2003 Prentice Hall, Inc.
All rights reserved.



76

Outline

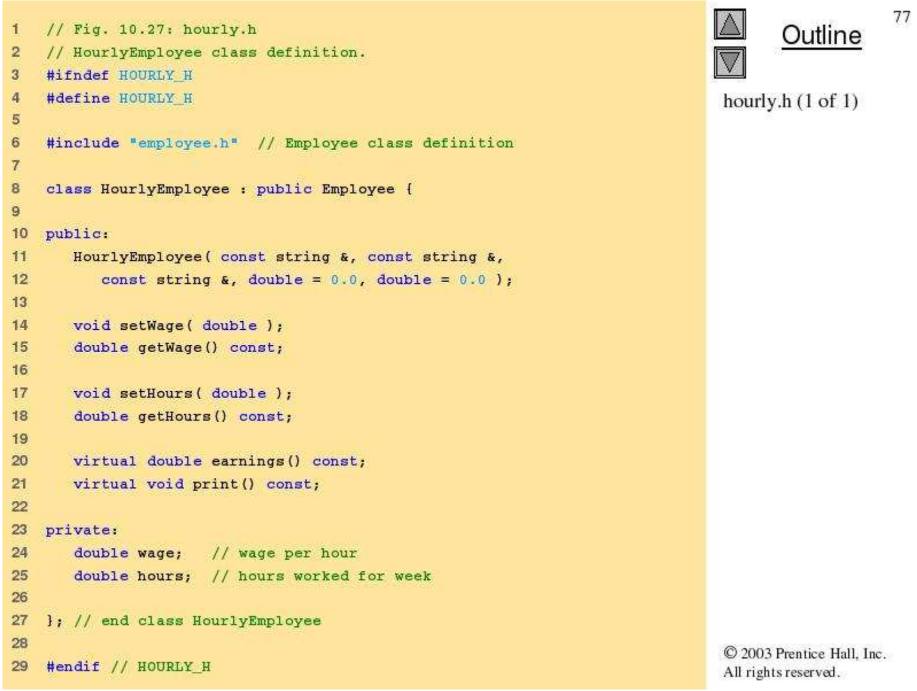
salaried.cpp
(2 of 2)

```
// calculate salaried employee's pay
double SalariedEmployee::earnings() const
{
   return getWeeklySalary();

} // end function earnings

// return salaried employee's salary
double SalariedEmployee::getWeeklySalary() const
{
   return weeklySalary;

} // end function getWeeklySalary

// print salaried employee's name
void SalariedEmployee::print() const
{
   cout << "\nsalaried employee: ";
   Employee::print();  // code reuse

} // end function print
```

Must implement pure virtual functions.

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 18: **SalariedEmployee** class implementation file.

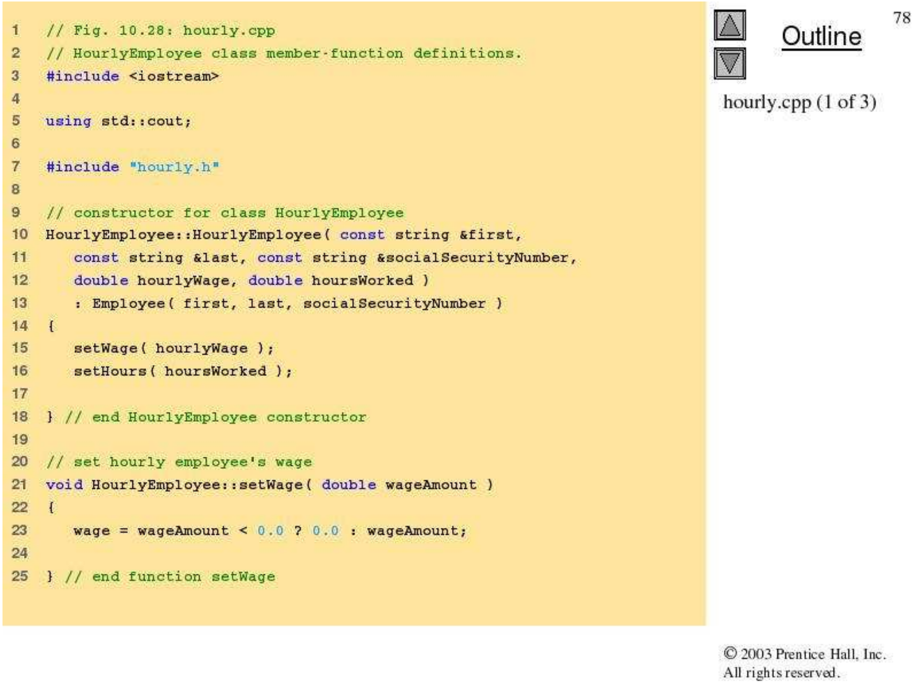


77

Outline

hourly.h (1 of 1)

```cpp
// Fig. 10.27: hourly.h
// HourlyEmployee class definition.
#ifndef HOURLY_H
#define HOURLY_H

#include "employee.h"  // Employee class definition

class HourlyEmployee : public Employee {

public:
   HourlyEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setWage( double );
   double getWage() const;

   void setHours( double );
   double getHours() const;

   virtual double earnings() const;
   virtual void print() const;

private:
   double wage;   // wage per hour
   double hours;  // hours worked for week

}; // end class HourlyEmployee

#endif // HOURLY_H
```



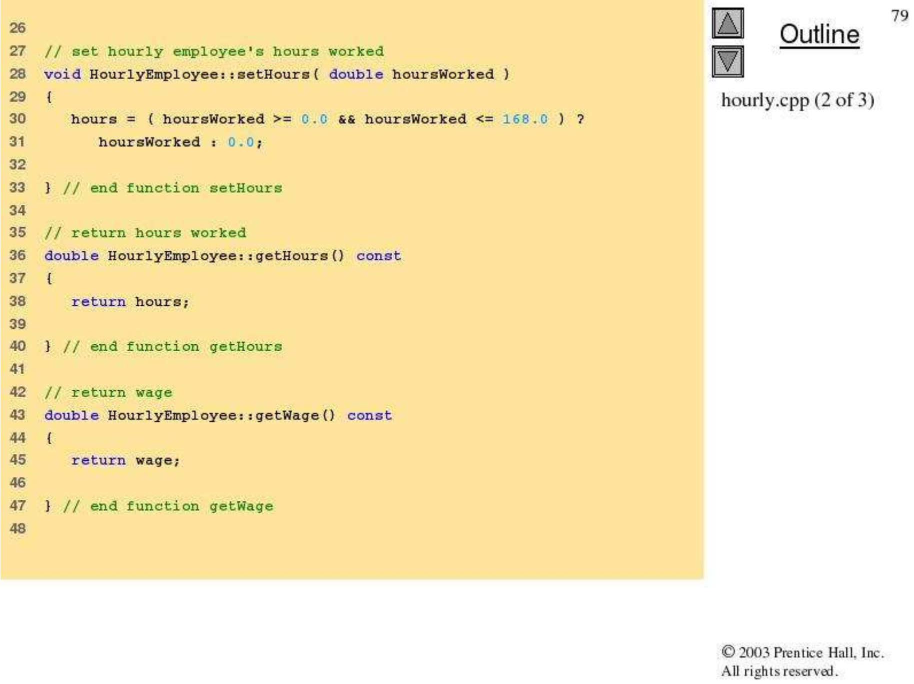
78

Outline

hourly.cpp (1 of 3)

```cpp
// Fig. 10.28: hourly.cpp
// HourlyEmployee class member-function definitions.
#include <iostream>

using std::cout;

#include "hourly.h"

// constructor for class HourlyEmployee
HourlyEmployee::HourlyEmployee( const string &first,
   const string &last, const string &socialSecurityNumber,
   double hourlyWage, double hoursWorked )
   : Employee( first, last, socialSecurityNumber )
{
   setWage( hourlyWage );
   setHours( hoursWorked );

} // end HourlyEmployee constructor

// set hourly employee's wage
void HourlyEmployee::setWage( double wageAmount )
{
   wage = wageAmount < 0.0 ? 0.0 : wageAmount;

} // end function setWage
```

Figure 19: **HourlyEmployee** class header file.



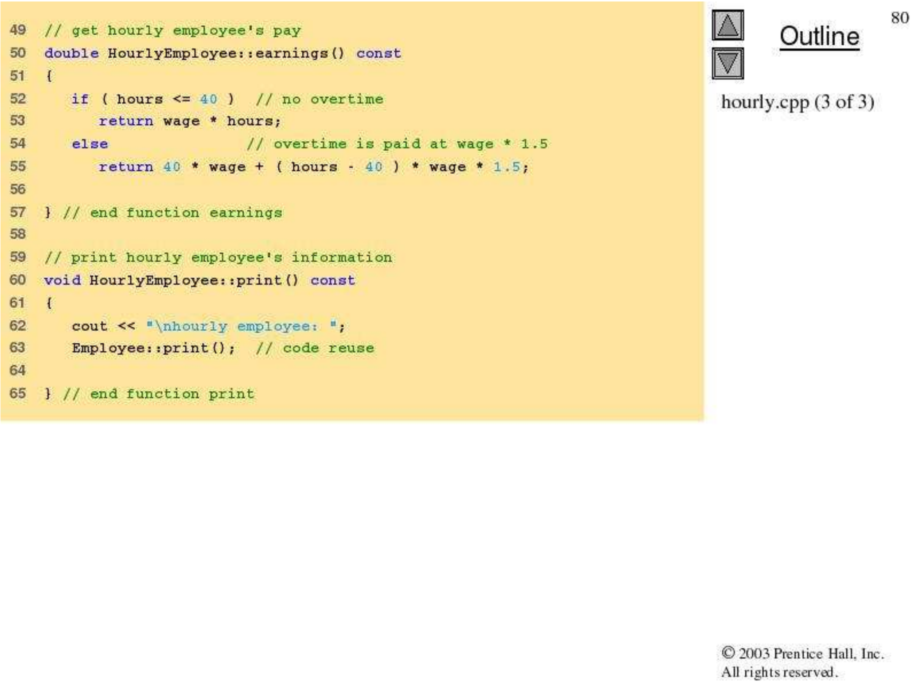
79

Outline

hourly.cpp (2 of 3)

```cpp

// set hourly employee's hours worked
void HourlyEmployee::setHours( double hoursWorked )
{
   hours = ( hoursWorked >= 0.0 && hoursWorked <= 168.0 ) ?
      hoursWorked : 0.0;

} // end function setHours

// return hours worked
double HourlyEmployee::getHours() const
{
   return hours;

} // end function getHours

// return wage
double HourlyEmployee::getWage() const
{
   return wage;

} // end function getWage

```

© 2003 Prentice Hall, Inc.
All rights reserved.



80

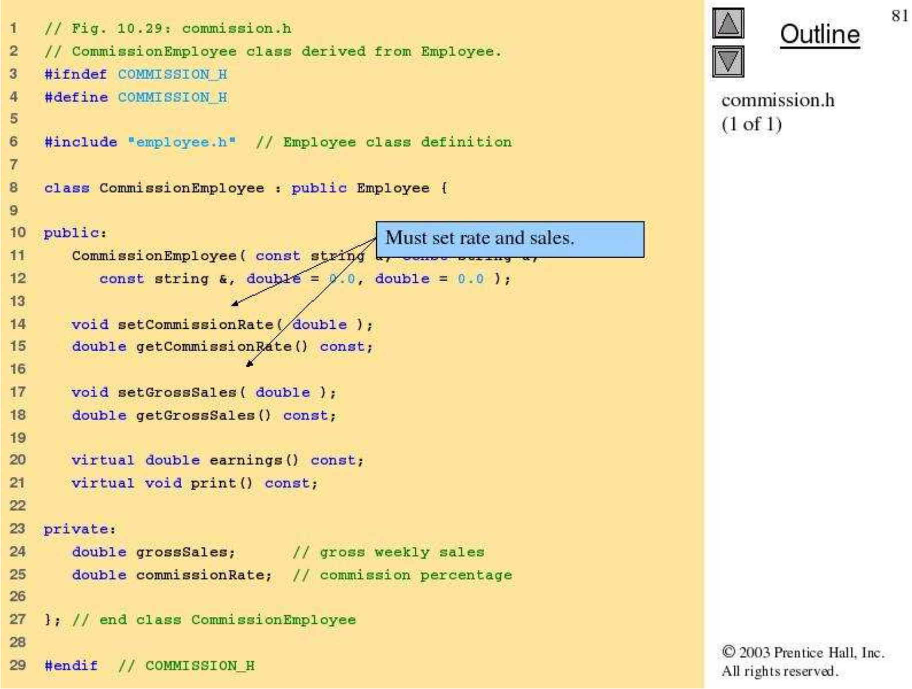
Outline

hourly.cpp (3 of 3)

```cpp
// get hourly employee's pay
double HourlyEmployee::earnings() const
{
   if ( hours <= 40 )  // no overtime
      return wage * hours;
   else                // overtime is paid at wage * 1.5
      return 40 * wage + ( hours - 40 ) * wage * 1.5;

} // end function earnings

// print hourly employee's information
void HourlyEmployee::print() const
{
   cout << "\nhourly employee: ";
   Employee::print();  // code reuse

} // end function print
```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 20: **HourlyEmployee** class implementation file.



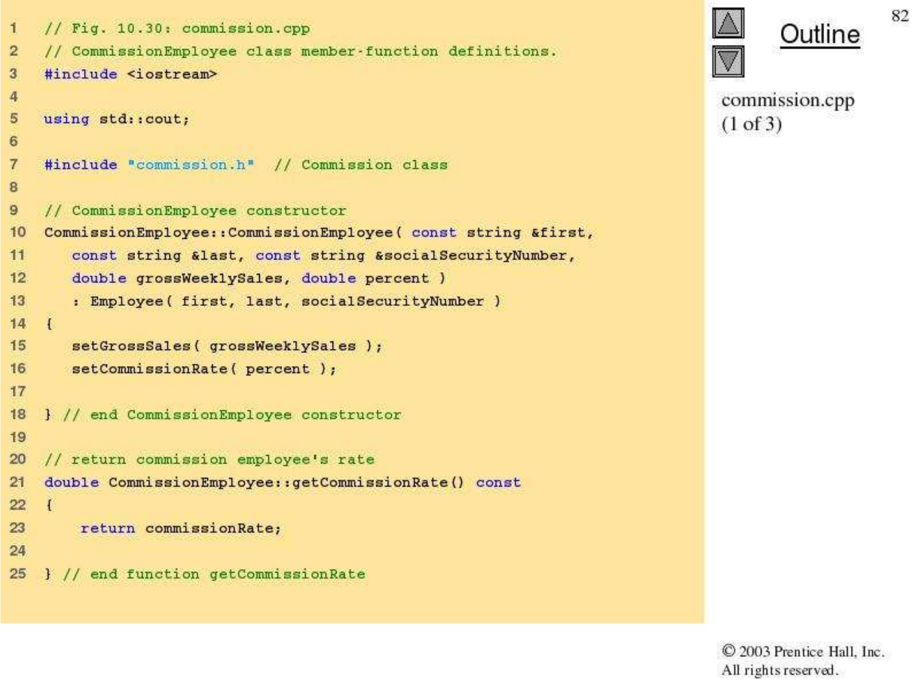
81

Outline

commission.h
(1 of 1)

```cpp
// Fig. 10.29: commission.h
// CommissionEmployee class derived from Employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include "employee.h"  // Employee class definition

class CommissionEmployee : public Employee {

public:
   CommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setCommissionRate( double );
   double getCommissionRate() const;

   void setGrossSales( double );
   double getGrossSales() const;

   virtual double earnings() const;
   virtual void print() const;

private:
   double grossSales;      // gross weekly sales
   double commissionRate;  // commission percentage

}; // end class CommissionEmployee

#endif  // COMMISSION_H
```

Must set rate and sales.



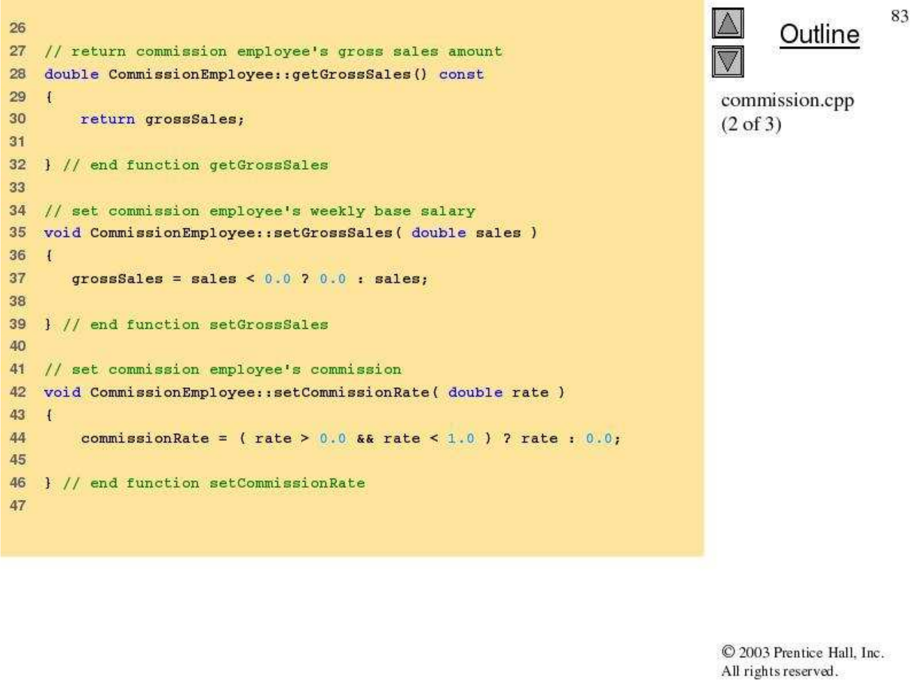
82

Outline

commission.cpp
(1 of 3)

```cpp
// Fig. 10.30: commission.cpp
// CommissionEmployee class member-function definitions.
#include <iostream>

using std::cout;

#include "commission.h"  // Commission class

// CommissionEmployee constructor
CommissionEmployee::CommissionEmployee( const string &first,
   const string &last, const string &socialSecurityNumber,
   double grossWeeklySales, double percent )
   : Employee( first, last, socialSecurityNumber )
{
   setGrossSales( grossWeeklySales );
   setCommissionRate( percent );

} // end CommissionEmployee constructor

// return commission employee's rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;

} // end function getCommissionRate
```

Figure 21: **CommissionEmployee** class header file.

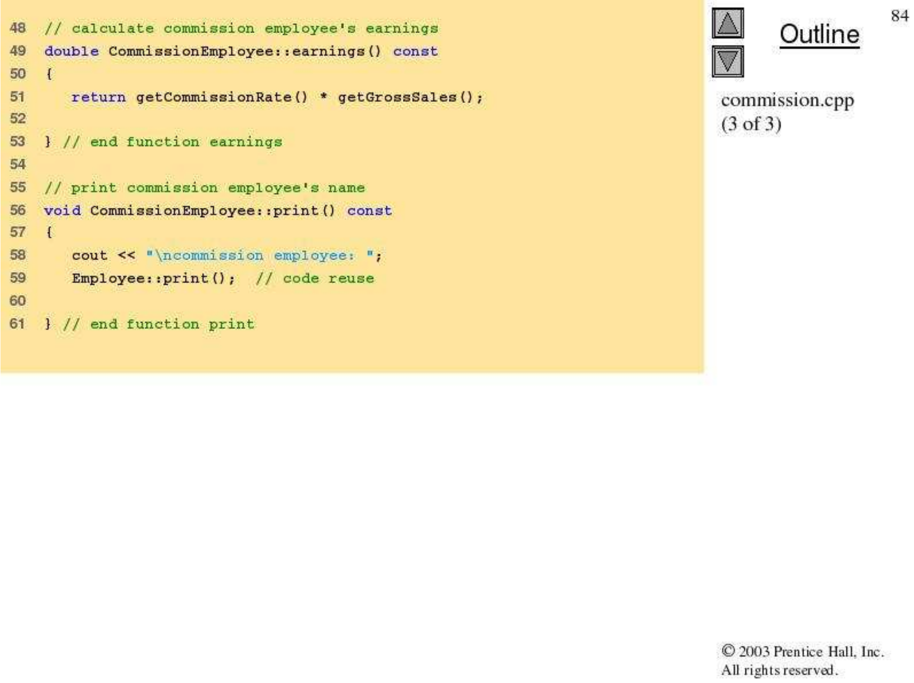


83

Outline

commission.cpp
(2 of 3)

```
// return commission employee's gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;

} // end function getGrossSales

// set commission employee's weekly base salary
void CommissionEmployee::setGrossSales( double sales )
{
   grossSales = sales < 0.0 ? 0.0 : sales;

} // end function setGrossSales

// set commission employee's commission
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;

} // end function setCommissionRate

```



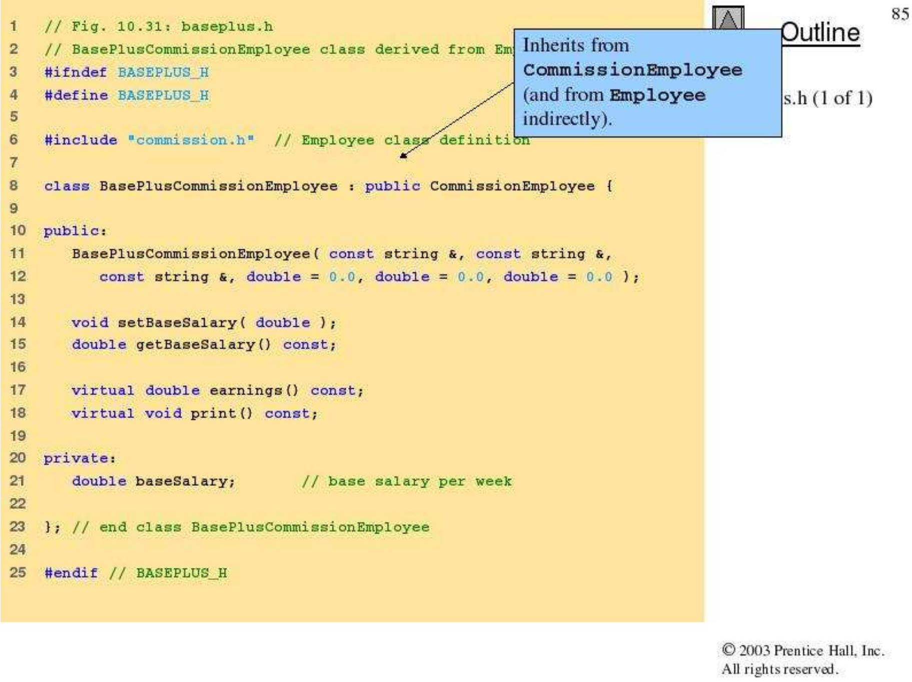
84

Outline

commission.cpp
(3 of 3)

```
// calculate commission employee's earnings
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();

} // end function earnings

// print commission employee's name
void CommissionEmployee::print() const
{
   cout << "\ncommission employee: ";
   Employee::print();  // code reuse

} // end function print
```

Figure 22: **CommissionEmployee** class implementation file.



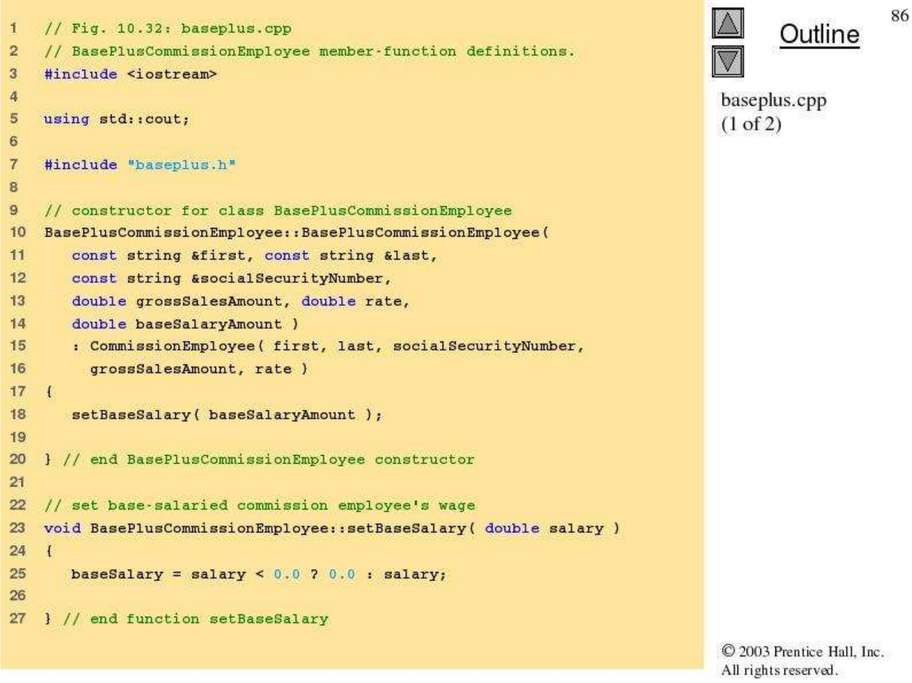
85

Outline

s.h (1 of 1)

```
// Fig. 10.31: baseplus.h
// BasePlusCommissionEmployee class derived from Em
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include "commission.h"  // Employee class definition

class BasePlusCommissionEmployee : public CommissionEmployee {

public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double );
   double getBaseSalary() const;

   virtual double earnings() const;
   virtual void print() const;

private:
   double baseSalary;       // base salary per week

}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

Inherits from **CommissionEmployee** (and from **Employee** indirectly).



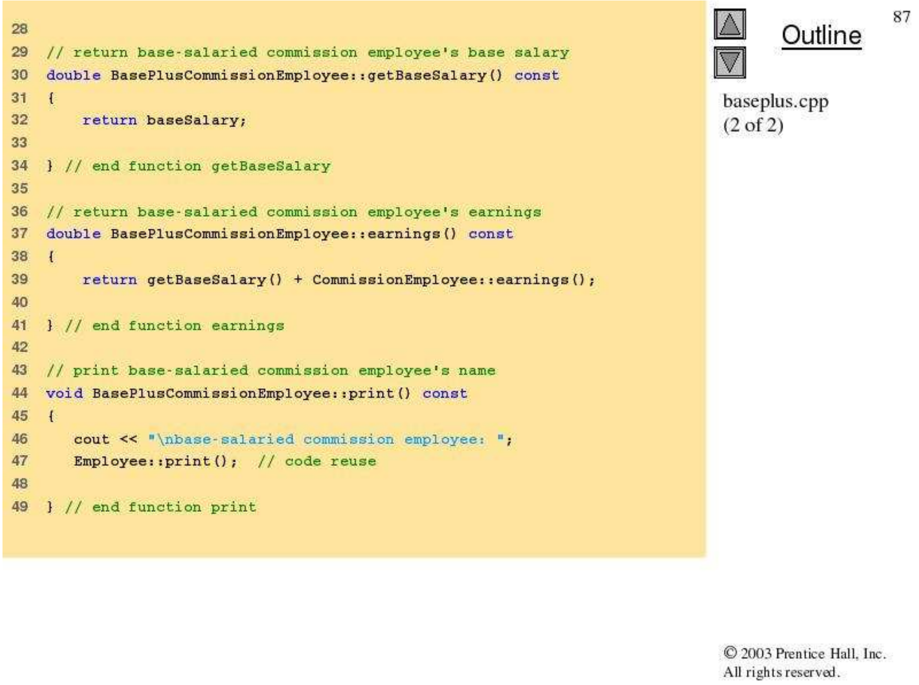
86

Outline

baseplus.cpp
(1 of 2)

```
// Fig. 10.32: baseplus.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iostream>

using std::cout;

#include "baseplus.h"

// constructor for class BasePlusCommissionEmployee
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last,
   const string &socialSecurityNumber,
   double grossSalesAmount, double rate,
   double baseSalaryAmount )
   : CommissionEmployee( first, last, socialSecurityNumber,
     grossSalesAmount, rate )
{
   setBaseSalary( baseSalaryAmount );

} // end BasePlusCommissionEmployee constructor

// set base-salaried commission employee's wage
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   baseSalary = salary < 0.0 ? 0.0 : salary;

} // end function setBaseSalary
```

Figure 23: **BasePlusCommissionEmployee** class header file.



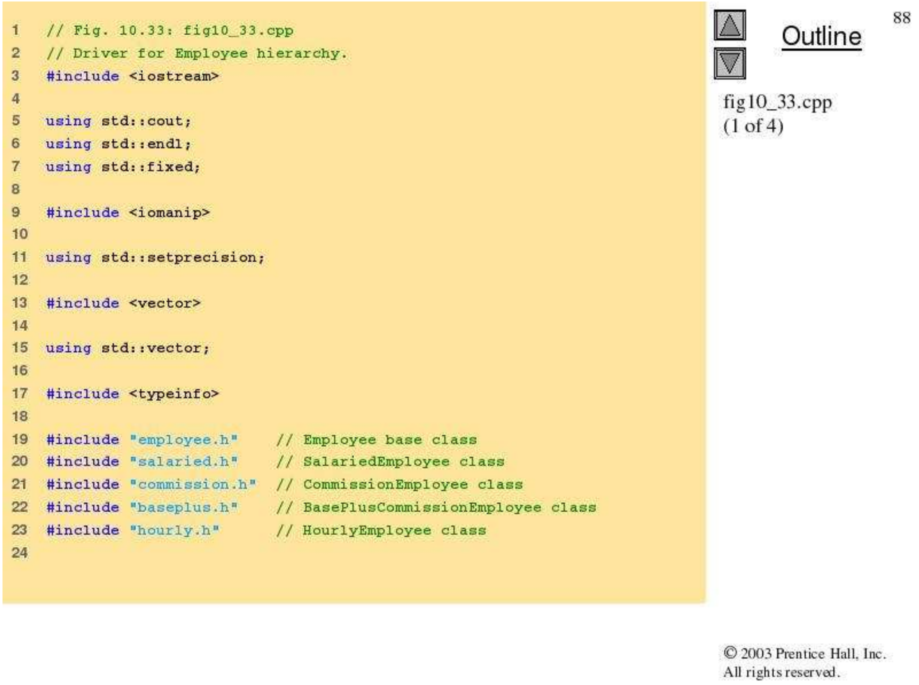
87

Outline

baseplus.cpp
(2 of 2)

```cpp

// return base-salaried commission employee's base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
   return baseSalary;

} // end function getBaseSalary

// return base-salaried commission employee's earnings
double BasePlusCommissionEmployee::earnings() const
{
   return getBaseSalary() + CommissionEmployee::earnings();

} // end function earnings

// print base-salaried commission employee's name
void BasePlusCommissionEmployee::print() const
{
   cout << "\nbase-salaried commission employee: ";
   Employee::print();  // code reuse

} // end function print
```

© 2003 Prentice Hall, Inc.
All rights reserved.



88

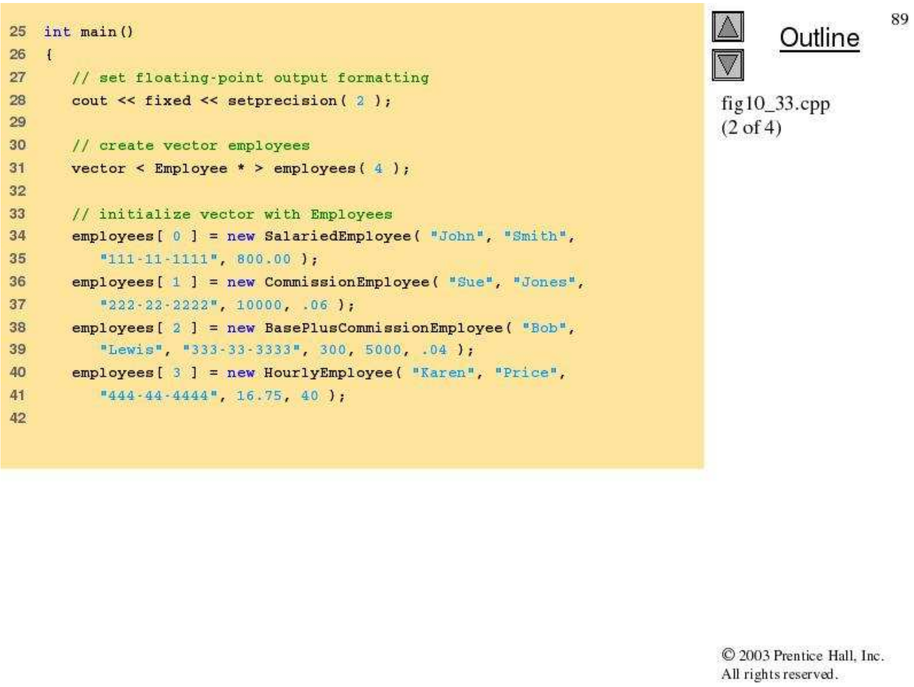
Outline

fig10_33.cpp
(1 of 4)

```cpp
// Fig. 10.33: fig10_33.cpp
// Driver for Employee hierarchy.
#include <iostream>

using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>

using std::setprecision;

#include <vector>

using std::vector;

#include <typeinfo>

#include "employee.h"     // Employee base class
#include "salaried.h"     // SalariedEmployee class
#include "commission.h"   // CommissionEmployee class
#include "baseplus.h"     // BasePlusCommissionEmployee class
#include "hourly.h"       // HourlyEmployee class

```

© 2003 Prentice Hall, Inc.
All rights reserved.

Figure 24: **BasePlusCommissionEmployee** class implementation file.

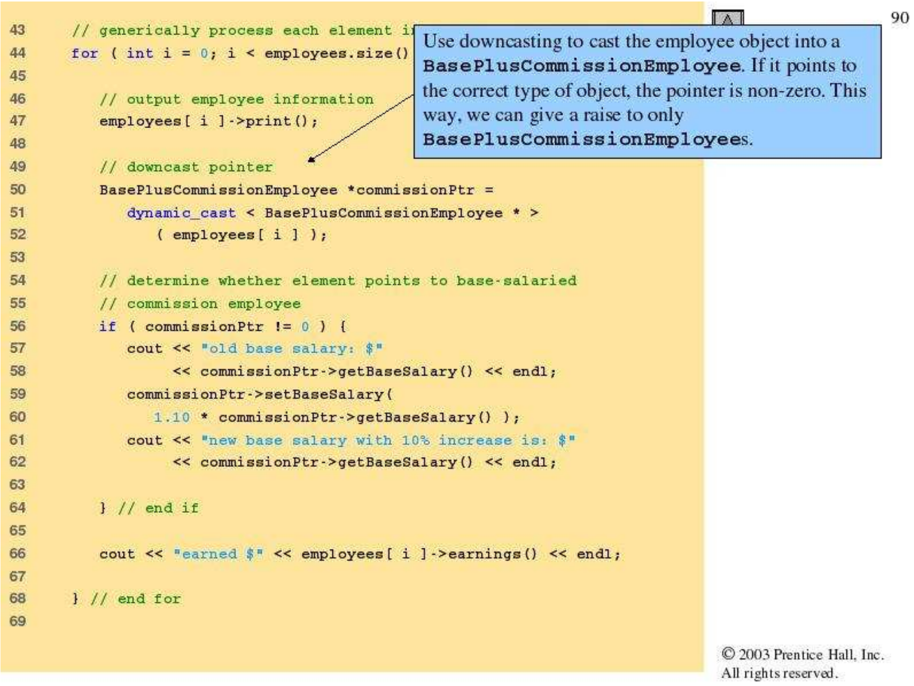


89

Outline

fig10_33.cpp
(2 of 4)

```
int main()
{
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // create vector employees
   vector < Employee * > employees( 4 );

   // initialize vector with Employees
   employees[ 0 ] = new SalariedEmployee( "John", "Smith",
      "111-11-1111", 800.00 );
   employees[ 1 ] = new CommissionEmployee( "Sue", "Jones",
      "222-22-2222", 10000, .06 );
   employees[ 2 ] = new BasePlusCommissionEmployee( "Bob",
      "Lewis", "333-33-3333", 300, 5000, .04 );
   employees[ 3 ] = new HourlyEmployee( "Karen", "Price",
      "444-44-4444", 16.75, 40 );

```

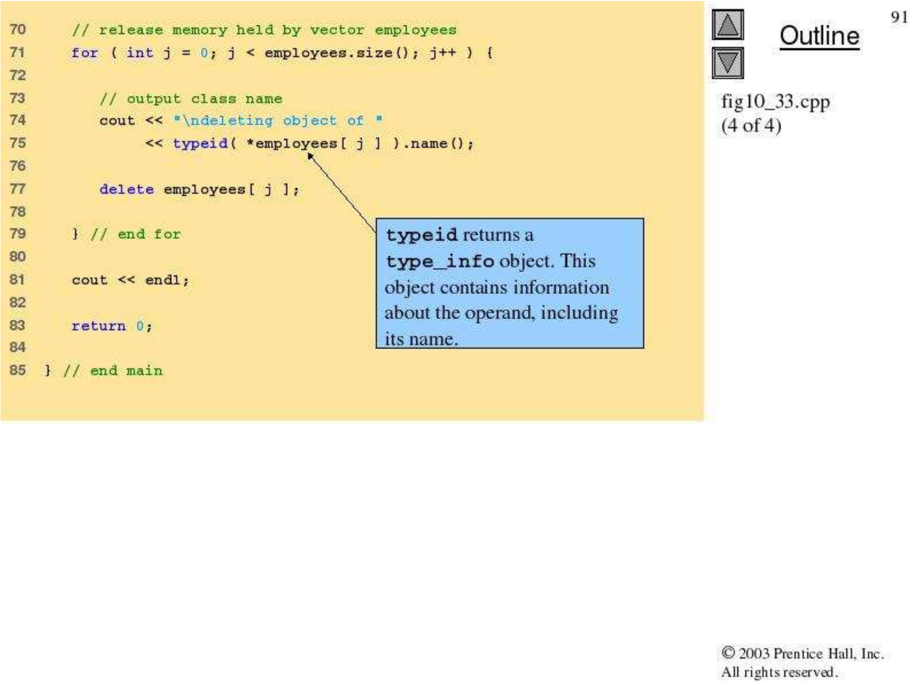


90

```
   // generically process each element i
   for ( int i = 0; i < employees.size()

      // output employee information
      employees[ i ]->print();

      // downcast pointer
      BasePlusCommissionEmployee *commissionPtr =
         dynamic_cast < BasePlusCommissionEmployee * >
            ( employees[ i ] );

      // determine whether element points to base-salaried
      // commission employee
      if ( commissionPtr != 0 ) {
         cout << "old base salary: $"
              << commissionPtr->getBaseSalary() << endl;
         commissionPtr->setBaseSalary(
            1.10 * commissionPtr->getBaseSalary() );
         cout << "new base salary with 10% increase is: $"
              << commissionPtr->getBaseSalary() << endl;

      } // end if

      cout << "earned $" << employees[ i ]->earnings() << endl;

   } // end for

```

Use downcasting to cast the employee object into a `BasePlusCommissionEmployee`. If it points to the correct type of object, the pointer is non-zero. This way, we can give a raise to only `BasePlusCommissionEmployee`s.

Figure 25: **Employee** class hierarchy driver program.(part 1 of 2)

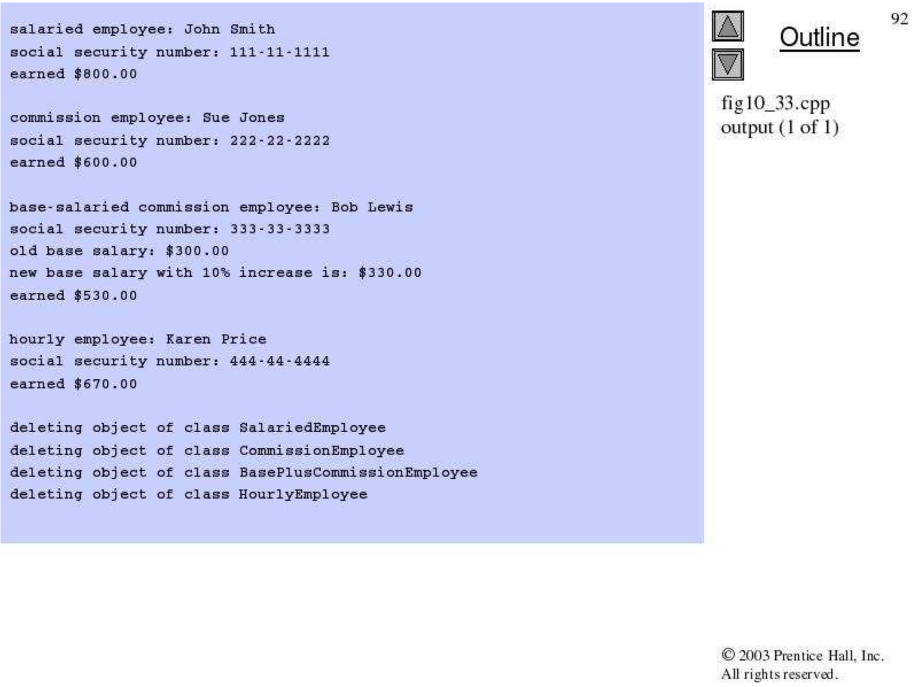


91

```
   // release memory held by vector employees
   for ( int j = 0; j < employees.size(); j++ ) {

      // output class name
      cout << "\ndeleting object of "
           << typeid( *employees[ j ] ).name();

      delete employees[ j ];

   } // end for

   cout << endl;

   return 0;

} // end main
```

`typeid` returns a `type_info` object. This object contains information about the operand, including its name.

fig10_33.cpp
(4 of 4)



92

```
salaried employee: John Smith
social security number: 111-11-1111
earned $800.00

commission employee: Sue Jones
social security number: 222-22-2222
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

hourly employee: Karen Price
social security number: 444-44-4444
earned $670.00

deleting object of class SalariedEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
deleting object of class HourlyEmployee
```

fig10_33.cpp
output (1 of 1)

Figure 26: **Employee** class hierarchy driver program.(part 2 of 2)