1 Object-Oriented Programming: Polymorphism

1.1 Introduction

- Polymorphism
  - ”Program in the general”
  - Treat objects in same class hierarchy as if all base class
  - Virtual functions and dynamic binding; will explain how polymorphism works
  - Makes programs extensible; new classes added easily, can still be processed
- In our examples
  - Use abstract base class **Shape**
    - Defines common interface (functionality)
    - **Point**, **Circle** and **Cylinder** inherit from **Shape**
  - Class **Employee** for a natural example

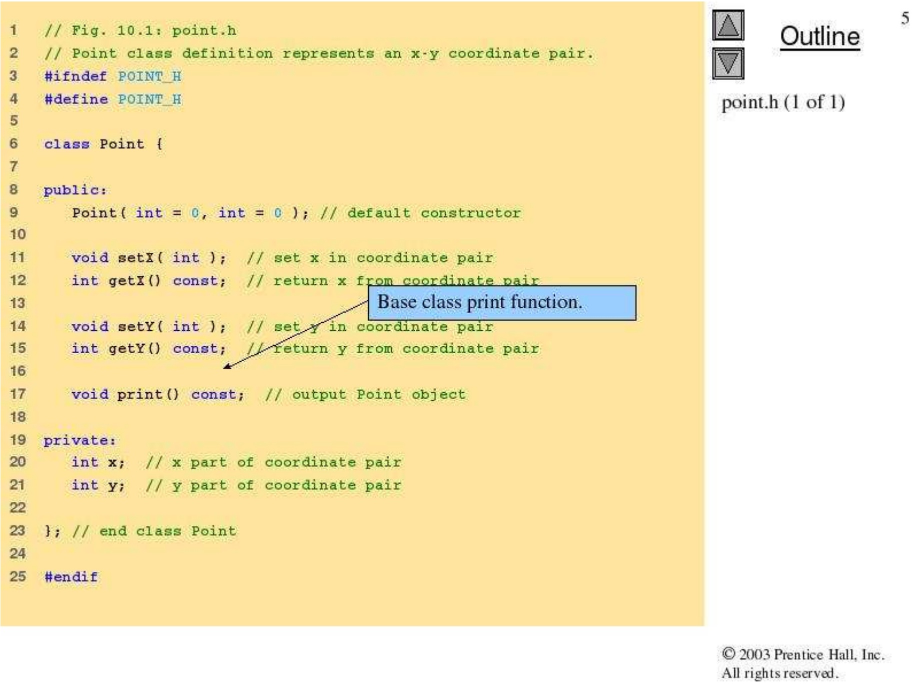
1.2 Relationships Among Objects in an Inheritance Hierarchy

- Previously (Section 9.4),
  - **Circle** inherited from **Point**
  - Manipulated **Point** and **Circle** objects using member functions
- Now
  - Invoke functions using base-class/derived-class pointers
  - Introduce **virtual** functions
- Key concept
  - Derived-class object can be treated as base-class object
    - ”is-a” relationship
    - Base class is not a derived class object

1.2.1 Invoking Base-Class Functions from Derived-Class Objects

Aim pointers (base, derived) at objects (base, derived)

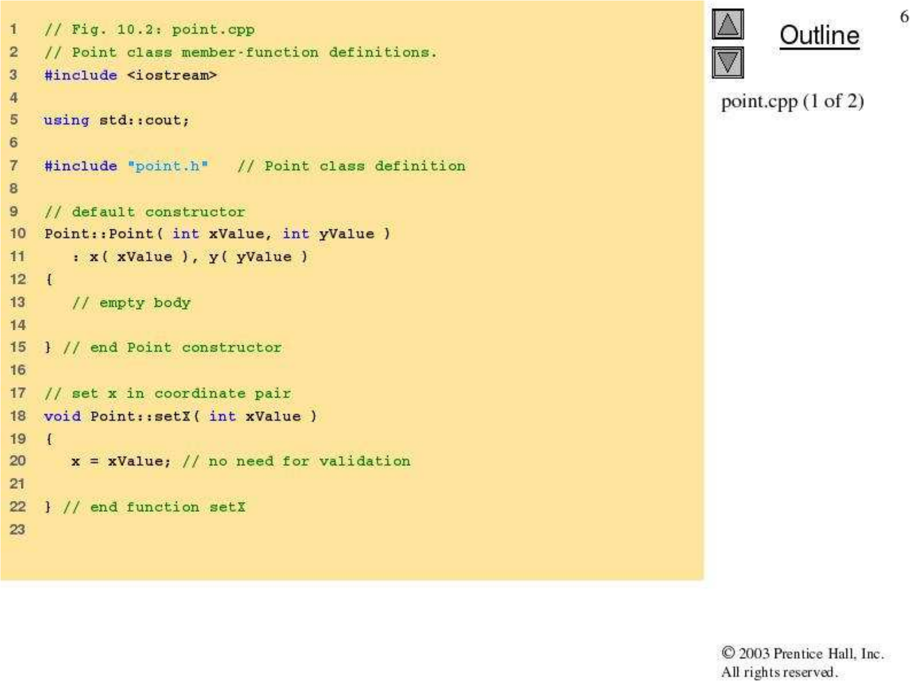
- Base pointer aimed at base object
- Derived pointer aimed at derived object; both straightforward
- Base pointer aimed at derived object
  - "is a" relationship; **Circle** "is a" **Point**
  - Will invoke base class functions
- Function call depends on the class of the pointer/handle
  - Does not depend on object to which it points
  - With **virtual** functions, this can be changed (more later)



```
// Fig. 10.1: point.h
// Point class definition represents an x-y coordinate pair.
#ifndef POINT_H
#define POINT_H

class Point {

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

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

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

   void print() const;  // output Point object

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

}; // end class Point

#endif
```

Base class print function.

Figure 1: **Point** class header file.

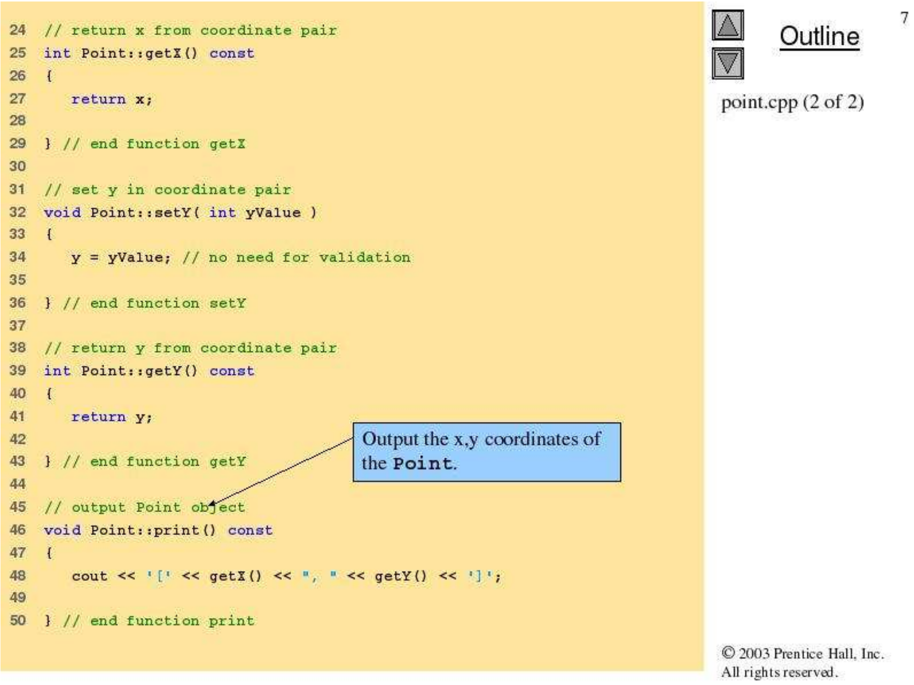


6

Outline

point.cpp (1 of 2)

```
// Fig. 10.2: 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

```



7

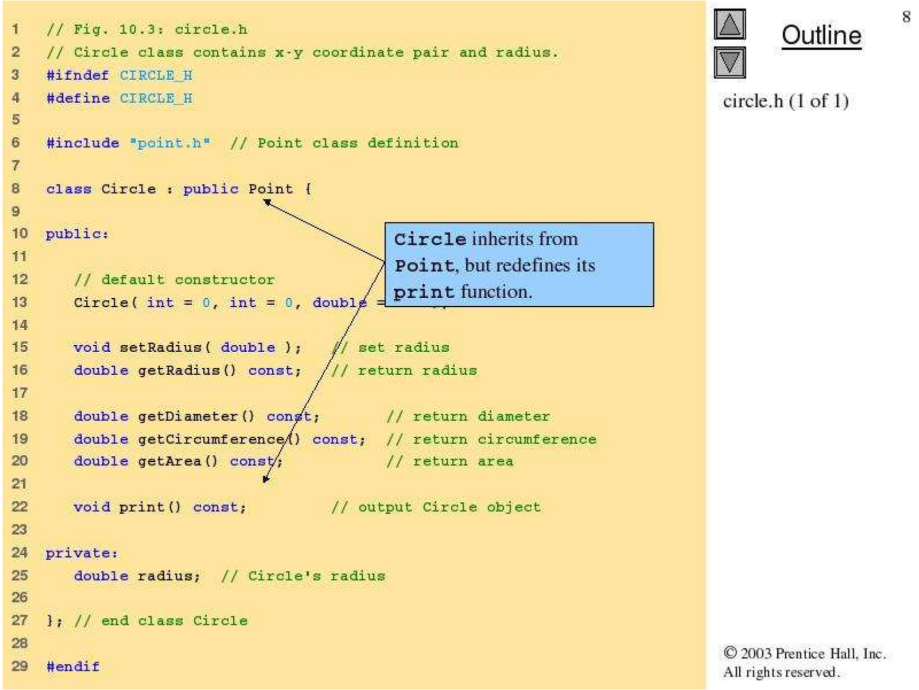
Outline

point.cpp (2 of 2)

```
// 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

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

} // end function print
```

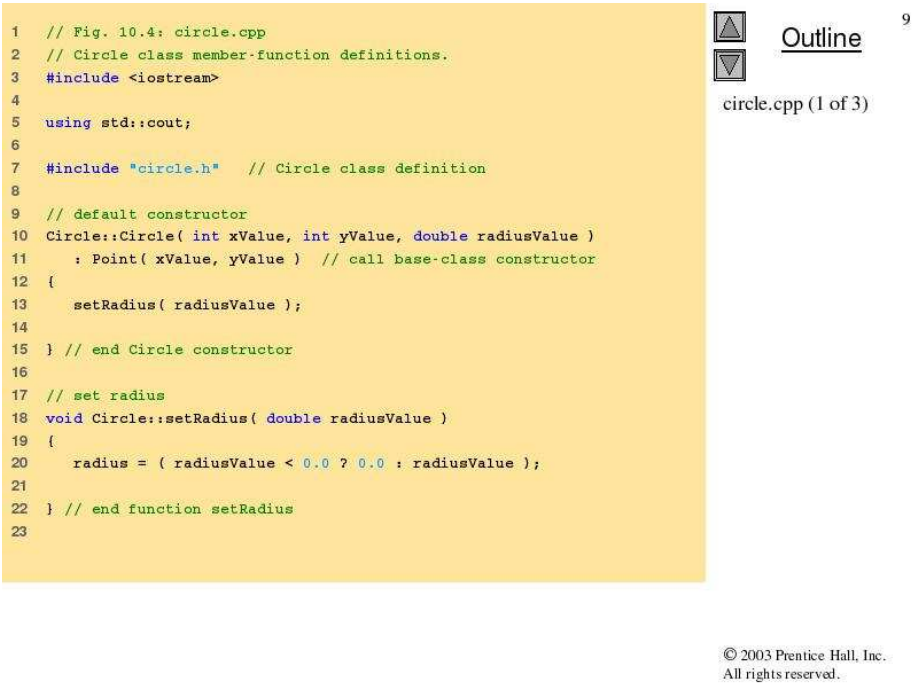
Output the x,y coordinates of the **Point**.

Figure 2: **Point** class represents an xy-coordinate pair.



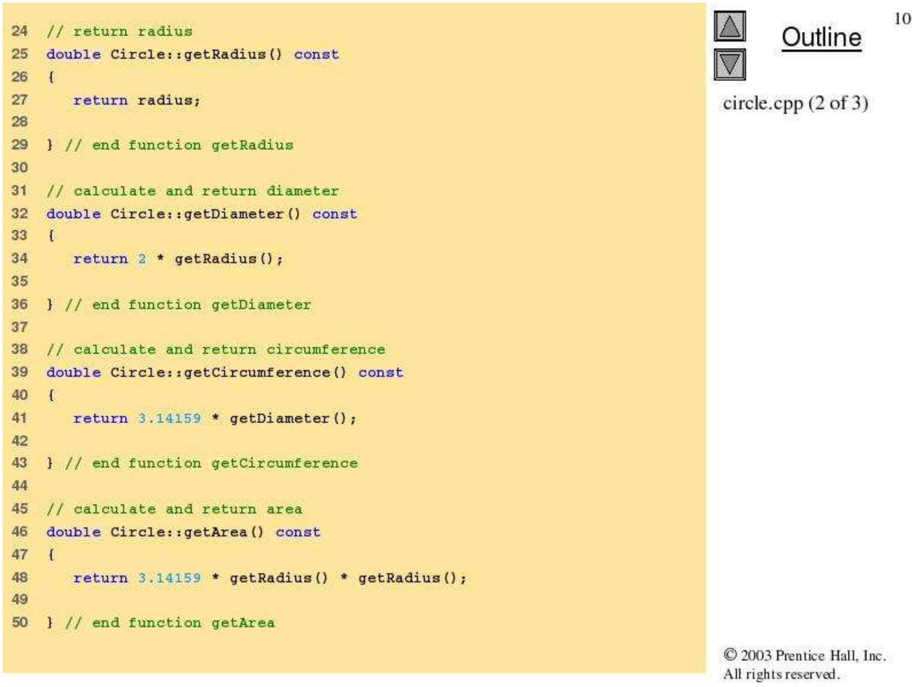
```
// Fig. 10.3: circle.h
// Circle class contains x-y coordinate pair and radius.
#ifndef CIRCLE_H
#define CIRCLE_H

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

class Circle : public Point {

public:

   // default constructor
   Circle( int = 0, int = 0, double =

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

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

   void print() const;         // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

Circle inherits from **Point**, but redefines its **print** function.



```
// Fig. 10.4: 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 3: **Circle** class header file.

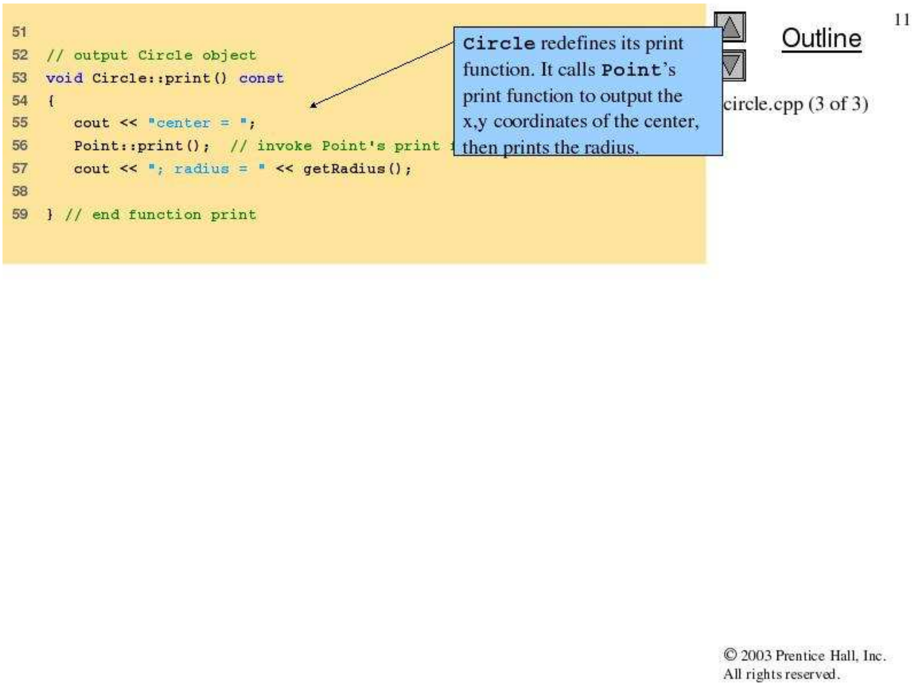


10

Outline

circle.cpp (2 of 3)

```cpp
// 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

// calculate and return area
double Circle::getArea() const
{
   return 3.14159 * getRadius() * getRadius();

} // end function getArea
```



11

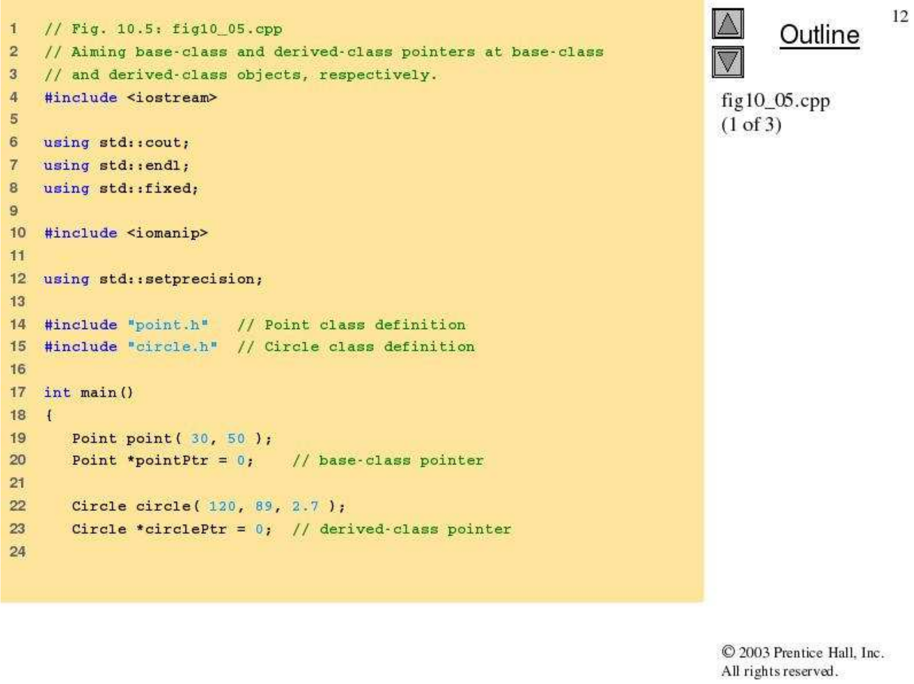
Outline

circle.cpp (3 of 3)

```cpp

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

} // end function print
```

Circle redefines its print function. It calls **Point**'s print function to output the x,y coordinates of the center, then prints the radius.

Figure 4: **Circle** class that inherits from class **Point**.

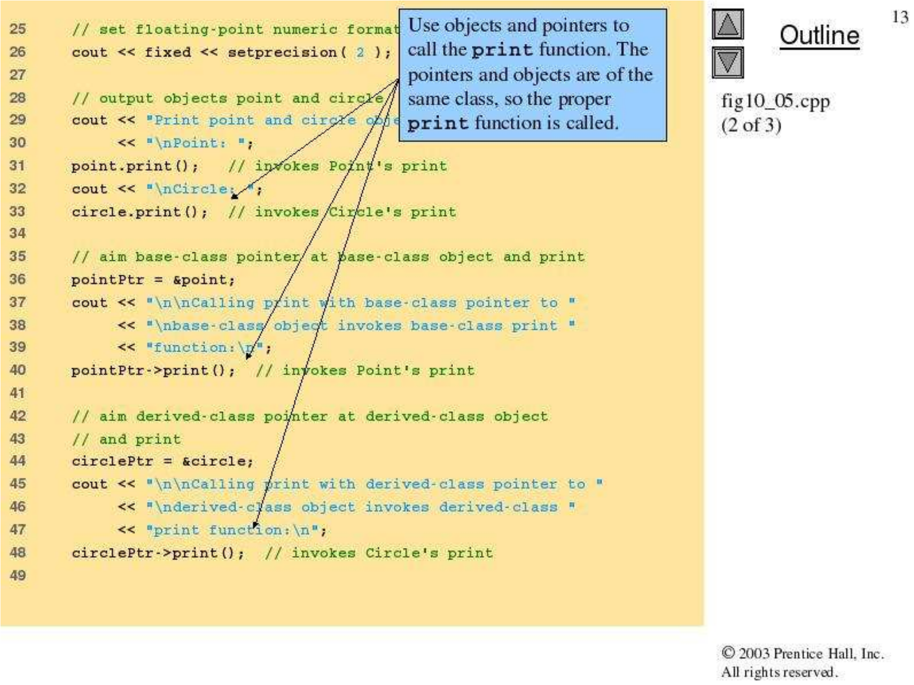


12

Outline

fig10_05.cpp
(1 of 3)

```
// Fig. 10.5: fig10_05.cpp
// Aiming base-class and derived-class pointers at base-class
// and derived-class objects, respectively.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

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

int main()
{
   Point point( 30, 50 );
   Point *pointPtr = 0;    // base-class pointer

   Circle circle( 120, 89, 2.7 );
   Circle *circlePtr = 0;  // derived-class pointer

```



13

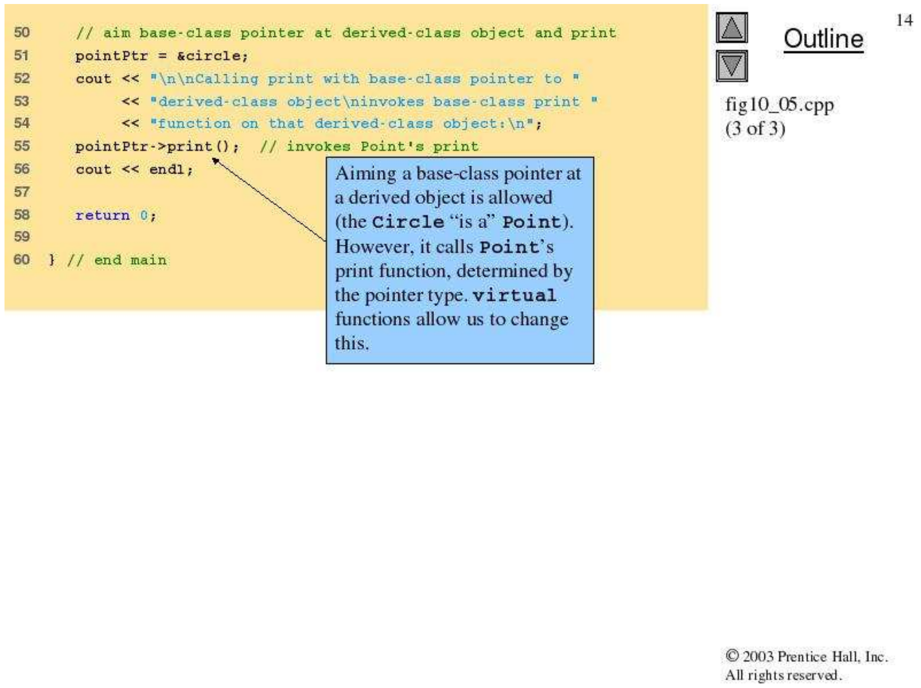
Outline

fig10_05.cpp
(2 of 3)

```
   // set floating-point numeric format
   cout << fixed << setprecision( 2 );

   // output objects point and circle
   cout << "Print point and circle obje
        << "\nPoint: ";
   point.print();   // invokes Point's print
   cout << "\nCircle: ";
   circle.print();  // invokes Circle's print

   // aim base-class pointer at base-class object and print
   pointPtr = &point;
   cout << "\n\nCalling print with base-class pointer to "
        << "\nbase-class object invokes base-class print "
        << "function:\n";
   pointPtr->print();  // invokes Point's print

   // aim derived-class pointer at derived-class object
   // and print
   circlePtr = &circle;
   cout << "\n\nCalling print with derived-class pointer to "
        << "\nderived-class object invokes derived-class "
        << "print function:\n";
   circlePtr->print();  // invokes Circle's print

```

Use objects and pointers to call the **print** function. The pointers and objects are of the same class, so the proper **print** function is called.

Figure 5: Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (part 1 of 2)



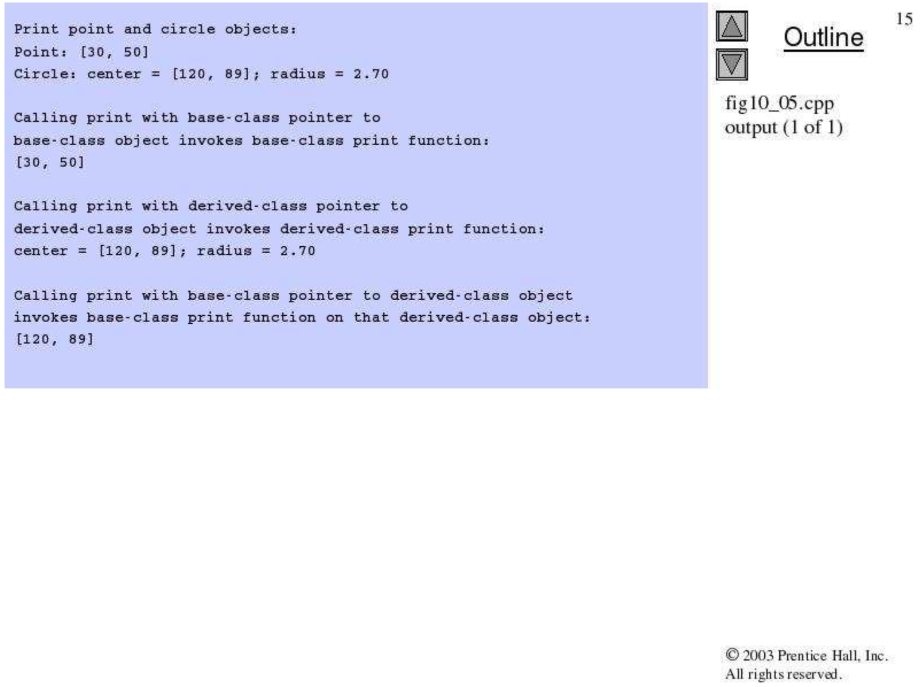
14

```
   // aim base-class pointer at derived-class object and print
   pointPtr = &circle;
   cout << "\n\nCalling print with base-class pointer to "
        << "derived-class object\ninvokes base-class print "
        << "function on that derived-class object:\n";
   pointPtr->print();  // invokes Point's print
   cout << endl;

   return 0;

} // end main
```

Aiming a base-class pointer at a derived object is allowed (the **Circle** “is a” **Point**). However, it calls **Point**’s print function, determined by the pointer type. **virtual** functions allow us to change this.

Outline

fig10_05.cpp
(3 of 3)



15

```
Print point and circle objects:
Point: [30, 50]
Circle: center = [120, 89]; radius = 2.70

Calling print with base-class pointer to
base-class object invokes base-class print function:
[30, 50]

Calling print with derived-class pointer to
derived-class object invokes derived-class print function:
center = [120, 89]; radius = 2.70

Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:
[120, 89]
```

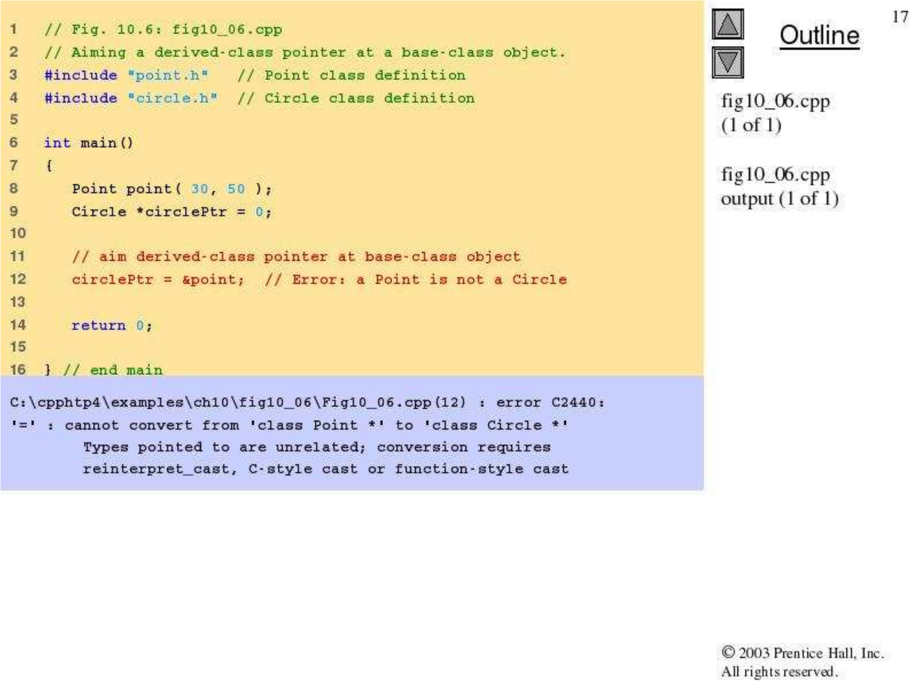
Outline

fig10_05.cpp
output (1 of 1)

Figure 6: Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (part 2 of 2)

1.2.2 Aiming Derived-Class Pointers at Base-Class Objects

- Previous example
  - Aimed base-class pointer at derived object; **Circle** "is a" **Point**
- Aim a derived-class pointer at a base-class object
  - Compiler error
    - No "is a" relationship
    - **Point** is not a **Circle**
    - **Circle** has data/functions that **Point** does not
      - **setRadius** (defined in **Circle**) not defined in **Point**
  - Can cast base-object"s address to derived-class pointer
    - Called downcasting (more in 10.9)
    - Allows derived-class functionality



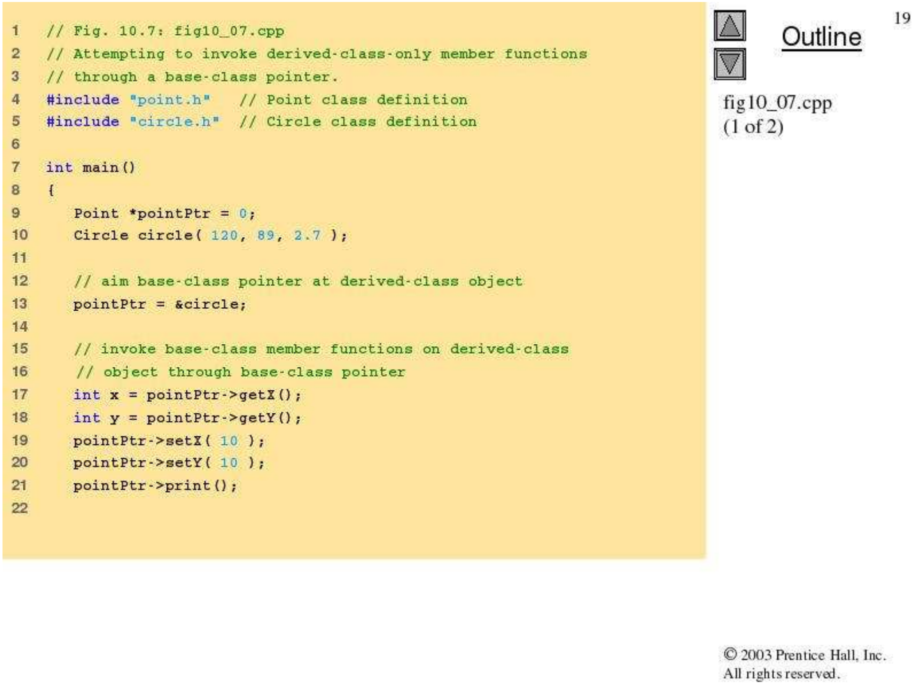
```
// Fig. 10.6: fig10_06.cpp
// Aiming a derived-class pointer at a base-class object.
#include "point.h"   // Point class definition
#include "circle.h"  // Circle class definition

int main()
{
   Point point( 30, 50 );
   Circle *circlePtr = 0;

   // aim derived-class pointer at base-class object
   circlePtr = &point;  // Error: a Point is not a Circle

   return 0;

} // end main
```

```
C:\cpphtp4\examples\ch10\fig10_06\Fig10_06.cpp(12) : error C2440:
'=' : cannot convert from 'class Point *' to 'class Circle *'
        Types pointed to are unrelated; conversion requires
        reinterpret_cast, C-style cast or function-style cast
```

Figure 7: Aiming a derived-class pointer at a base-class object.

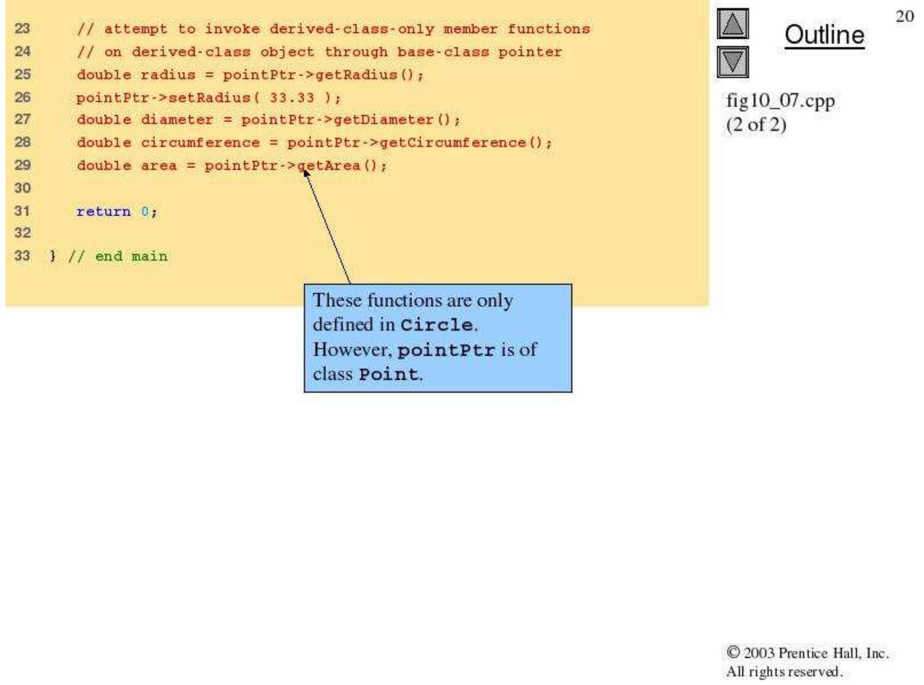
1.2.3 Derived-Class Member-Function Calls via Base-Class Pointers

- Handle (pointer/reference)
  - Base-pointer can aim at derived-object; but can only call base-class functions
  - Calling derived-class functions is a compiler error; functions not defined in base-class
- Common theme
  - Data type of pointer/reference determines functions it can call



```
// Fig. 10.7: fig10_07.cpp
// Attempting to invoke derived-class-only member functions
// through a base-class pointer.
#include "point.h"   // Point class definition
#include "circle.h"  // Circle class definition

int main()
{
   Point *pointPtr = 0;
   Circle circle( 120, 89, 2.7 );

   // aim base-class pointer at derived-class object
   pointPtr = &circle;

   // invoke base-class member functions on derived-class
   // object through base-class pointer
   int x = pointPtr->getX();
   int y = pointPtr->getY();
   pointPtr->setX( 10 );
   pointPtr->setY( 10 );
   pointPtr->print();

```

Figure 8: Attempting to invoke derived-class-only functions via a base-class pointer. (part 1 of 2)



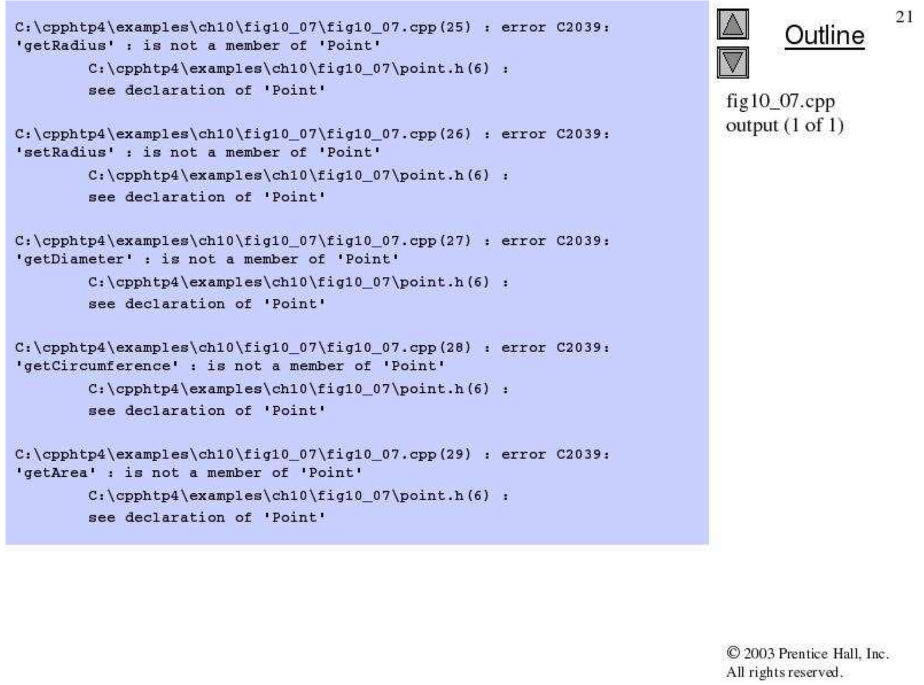
20

Outline

fig10_07.cpp
(2 of 2)

```
23      // attempt to invoke derived-class-only member functions
24      // on derived-class object through base-class pointer
25      double radius = pointPtr->getRadius();
26      pointPtr->setRadius( 33.33 );
27      double diameter = pointPtr->getDiameter();
28      double circumference = pointPtr->getCircumference();
29      double area = pointPtr->getArea();
30
31      return 0;
32
33   } // end main
```

These functions are only defined in `Circle`. However, `pointPtr` is of class `Point`.



21

Outline

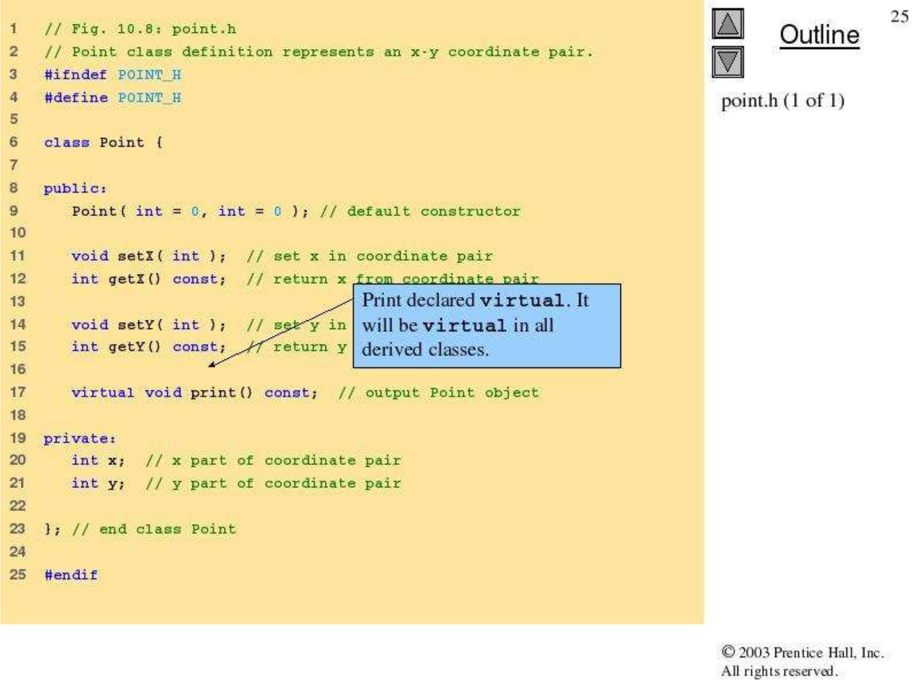
fig10_07.cpp
output (1 of 1)

```
C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(25) : error C2039:
'getRadius' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(26) : error C2039:
'setRadius' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(27) : error C2039:
'getDiameter' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(28) : error C2039:
'getCircumference' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'

C:\cpphtp4\examples\ch10\fig10_07\fig10_07.cpp(29) : error C2039:
'getArea' : is not a member of 'Point'
        C:\cpphtp4\examples\ch10\fig10_07\point.h(6) :
        see declaration of 'Point'
```

Figure 9: Attempting to invoke derived-class-only functions via a base-class pointer. (part 2 of 2)

1.2.4 Virtual Functions

- Typically, pointer-class determines functions
- virtual functions; object (not pointer) determines function called
- Why useful?
  - Suppose **Circle**, **Triangle**, **Rectangle** derived from **Shape**; each has own **draw** function
  - To draw any shape
    - Have base class **Shape** pointer, call **draw**
    - Program determines proper **draw** function at run time (dynamically)
    - Treat all shapes generically
- Declare draw as virtual in base class
  - Override **draw** in each derived class; like redefining, but new function must have same signature
  - If function declared **virtual**, can only be overridden
    - **virtual void draw() const;**
    - Once declared **virtual**, **virtual** in all derived classes; good practice to explicitly declare **virtual**
- Dynamic binding
  - Choose proper function to call at run time
  - Only occurs off pointer handles; if function called from object, uses that object"s definition
- Example
  - Redo **Point**, **Circle** example with **virtual** functions
  - Base-class pointer to derived-class object; will call derived-class function
- Polymorphism
  - Same message, "print", given to many objects; all through a base pointer
  - Message takes on "many forms"



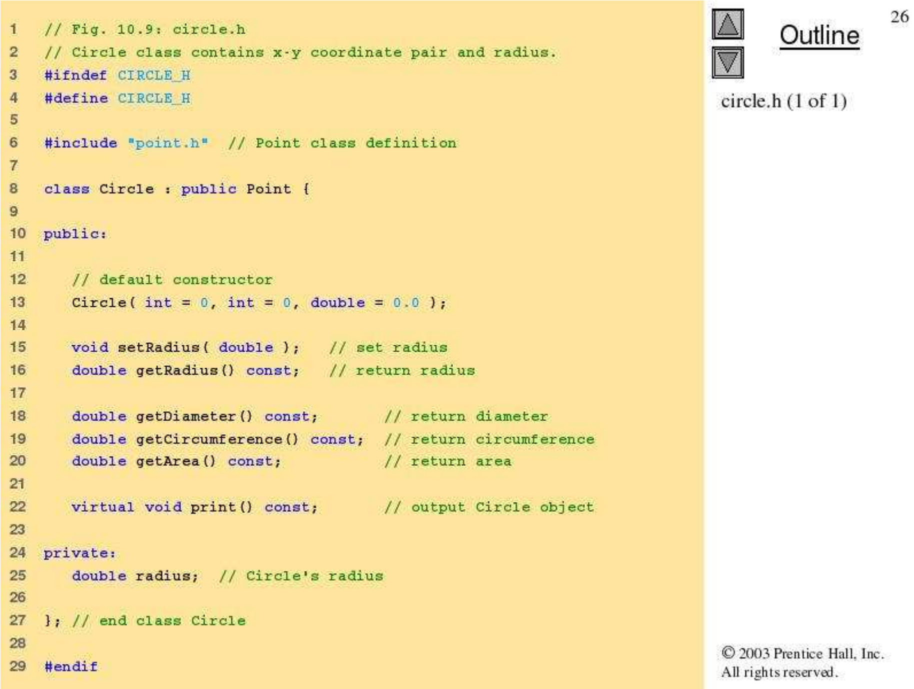
25

Outline

point.h (1 of 1)

```cpp
// Fig. 10.8: point.h
// Point class definition represents an x-y coordinate pair.
#ifndef POINT_H
#define POINT_H

class Point {

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

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in
   int getY() const;  // return y

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

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

}; // end class Point

#endif
```

Print declared **`virtual`**. It will be **`virtual`** in all derived classes.



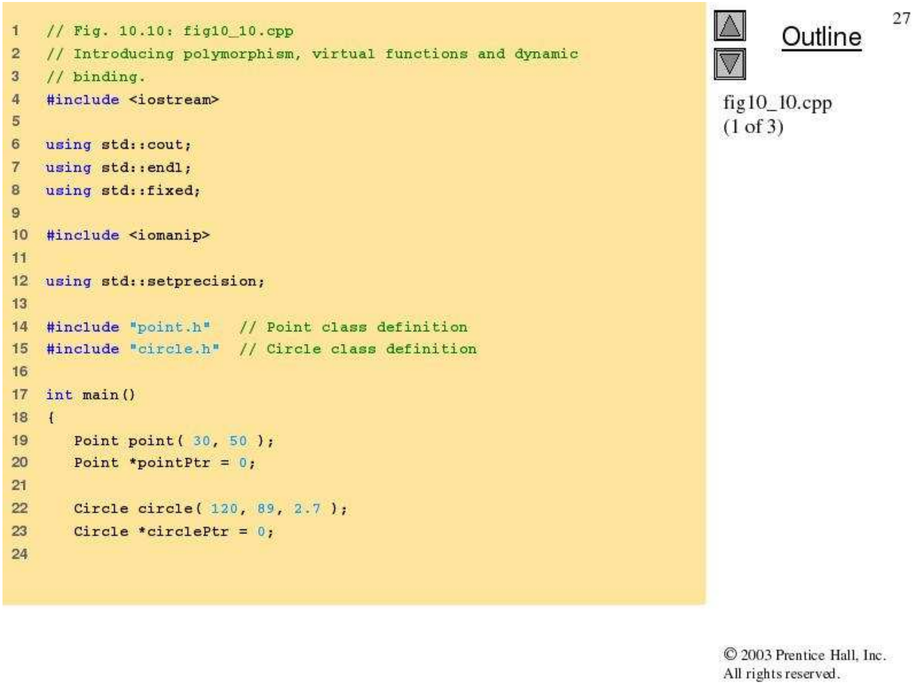
26

Outline

circle.h (1 of 1)

```cpp
// Fig. 10.9: 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
   double getArea() const;           // return area

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

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

Figure 10: **Point** class header file declares **print** function as **virtual** (upper) and **Circle** class header file declares **print** function as **virtual**.

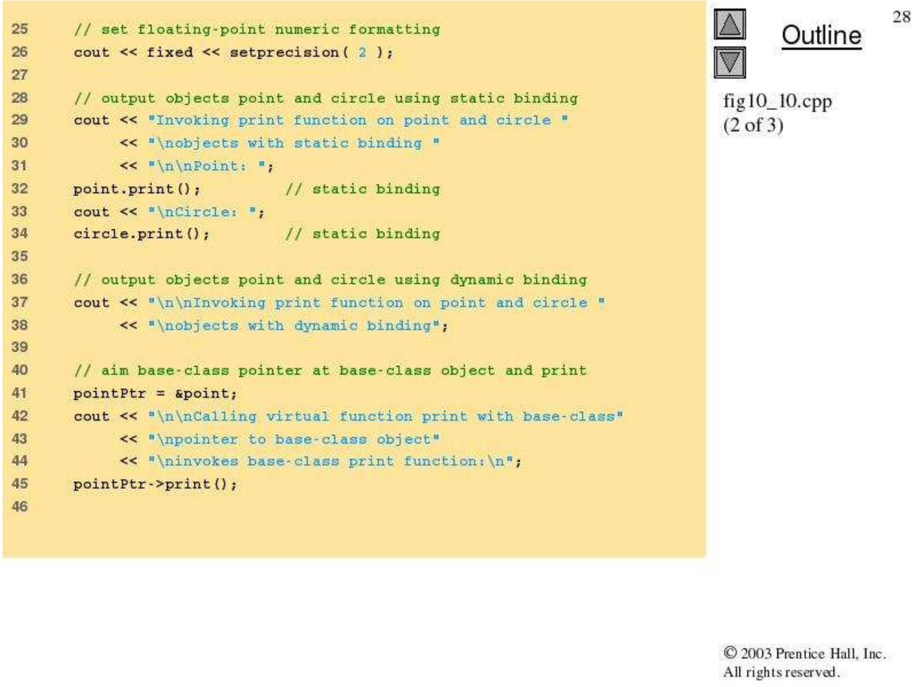


27

Outline

fig10_10.cpp
(1 of 3)

```cpp
// Fig. 10.10: fig10_10.cpp
// Introducing polymorphism, virtual functions and dynamic
// binding.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

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

int main()
{
   Point point( 30, 50 );
   Point *pointPtr = 0;

   Circle circle( 120, 89, 2.7 );
   Circle *circlePtr = 0;

```



28

Outline

fig10_10.cpp
(2 of 3)

```cpp
   // set floating-point numeric formatting
   cout << fixed << setprecision( 2 );

   // output objects point and circle using static binding
   cout << "Invoking print function on point and circle "
        << "\nobjects with static binding "
        << "\n\nPoint: ";
   point.print();         // static binding
   cout << "\nCircle: ";
   circle.print();        // static binding

   // output objects point and circle using dynamic binding
   cout << "\n\nInvoking print function on point and circle "
        << "\nobjects with dynamic binding";

   // aim base-class pointer at base-class object and print
   pointPtr = &point;
   cout << "\n\nCalling virtual function print with base-class"
        << "\npointer to base-class object"
        << "\ninvokes base-class print function:\n";
   pointPtr->print();

```

Figure 11: Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object. (part 1 of 2)

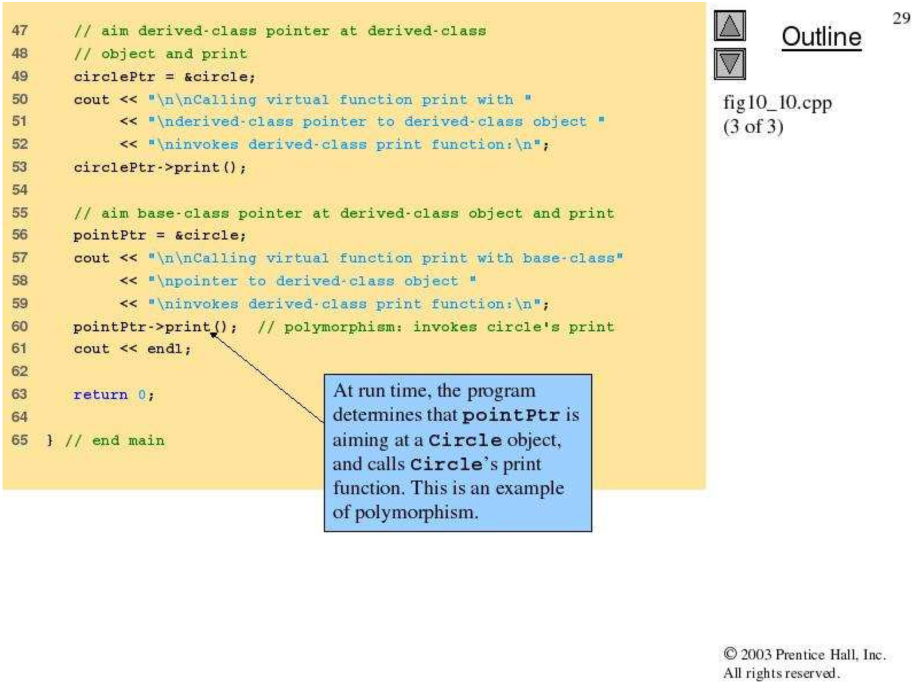


fig10_10.cpp
(3 of 3)

```
   // aim derived-class pointer at derived-class
   // object and print
   circlePtr = &circle;
   cout << "\n\nCalling virtual function print with "
        << "\nderived-class pointer to derived-class object "
        << "\ninvokes derived-class print function:\n";
   circlePtr->print();

   // aim base-class pointer at derived-class object and print
   pointPtr = &circle;
   cout << "\n\nCalling virtual function print with base-class"
        << "\npointer to derived-class object "
        << "\ninvokes derived-class print function:\n";
   pointPtr->print();  // polymorphism: invokes circle's print
   cout << endl;

   return 0;

} // end main
```

At run time, the program determines that `pointPtr` is aiming at a `Circle` object, and calls `Circle`'s print function. This is an example of polymorphism.

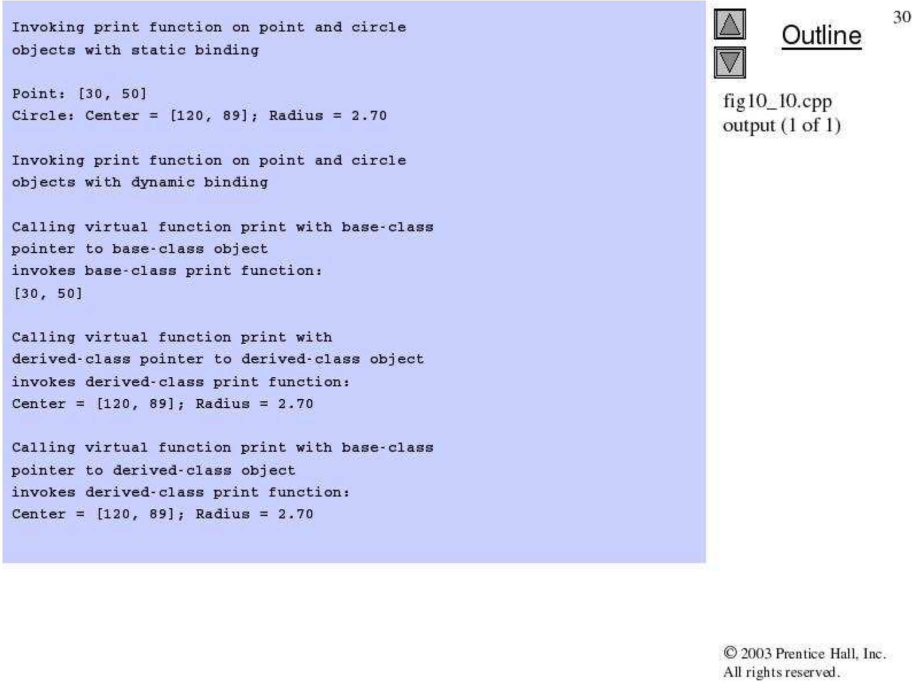


fig10_10.cpp
output (1 of 1)

```
Invoking print function on point and circle
objects with static binding

Point: [30, 50]
Circle: Center = [120, 89]; Radius = 2.70

Invoking print function on point and circle
objects with dynamic binding

Calling virtual function print with base-class
pointer to base-class object
invokes base-class print function:
[30, 50]

Calling virtual function print with
derived-class pointer to derived-class object
invokes derived-class print function:
Center = [120, 89]; Radius = 2.70

Calling virtual function print with base-class
pointer to derived-class object
invokes derived-class print function:
Center = [120, 89]; Radius = 2.70
```

Figure 12: Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object. (part 2 of 2)

- Summary
  - Base-pointer to base-object, derived-pointer to derived; straightforward
  - Base-pointer to derived object; can only call base-class functions
  - Derived-pointer to base-object
    - Compiler error
    - Allowed if explicit cast made (more in section 10.9)

1.3 Polymorphism Examples

- Suppose **Rectangle** derives from **Quadrilateral**
  - **Rectangle** more specific **Quadrilateral**
  - Any operation on **Quadrilateral** can be done on **Rectangle** (i.e., perimeter, area)
- Suppose designing video game
  - Base class **SpaceObject**
    - Derived **Martian**, **SpaceShip**, **LaserBeam**
    - Base function **draw**
  - To refresh screen
    - Screen manager has **vector** of base-class pointers to objects
    - Send **draw** message to each object
    - Same message has ”many forms” of results

1.4 Type Fields and switch Structures

- One way to determine object’s class
  - Give base class an attribute; **shapeType** in class **Shape**
  - Use **switch** to call proper **print** function
- Many problems
  - May forget to test for case in **switch**
  - If add/remove a class, must update **switch** structures; Time consuming and error prone
- Better to use polymorphism
  - Less branching logic, simpler programs, less debugging

1.5 Abstract Classes

- Abstract classes
  - Sole purpose: to be a base class (called abstract base classes)
  - Incomplete; derived classes fill in "missing pieces"
  - Cannot make objects from abstract class; however, can have pointers and references
- Concrete classes
  - Can instantiate objects
  - Implement all functions they define
  - Provide specifics
- Abstract classes not required, but helpful
- To make a class abstract
  - Need one or more "pure" virtual functions
    - Declare function with initializer of 0
    - **virtual void draw() const = 0;**
  - Regular virtual functions; have implementations, overriding is optional
  - Pure virtual functions; no implementation, must be overridden
  - Abstract classes can have data and concrete functions; required to have one or more pure virtual functions
- Abstract base class pointers; useful for polymorphism
- Application example
  - Abstract class **Shape**; defines **draw** as pure virtual function
  - **Circle**, **Triangle**, **Rectangle** derived from **Shape**; each must implement **draw**
  - Screen manager knows that each object can draw itself
- Iterators (more Chapter 21)
  - Walk through elements in **vector**/array
  - Use base-class pointer to send **draw** message to each