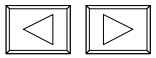
# Capitolo 10- Virtual Functions and Polymorphism

**Outline**

**10.1 Introduction**
**10.2 Type Fields and `switch` Statements**
**10.3 Virtual Functions**
**10.4 Abstract Base Classes and Concrete Classes**
**10.5 Polymorphism**
**10.6 Case Study: A Payroll System Using Polymorphism**
**10.7 New Classes and Dynamic Binding**
**10.8 Virtual Destructors**
**10.9 Case Study: Inheriting Interface and Implementation**
**10.10 Polymorphism, `virtual` Functions and Dynamic Binding "Under the Hood"**

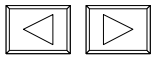




# 10.1 Introduction

- **`virtual`** functions and polymorphism
  - Design and implement systems that are more easily extensible
  - Programs written to generically process objects of all existing classes in a hierarchy

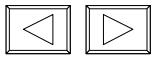




# 10.2 Type Fields and `switch` Statements

- **`switch`** statement
  - Take an action on a object based on its type
  - A switch structure could determine which **`print`** function to call based on which type in a hierarchy of shapes
- Problems with **`switch`**
  - Programmer may forget to test all possible cases in a **`switch`**
    - Tracking this down can be time consuming and prone to error
  - **`virtual`** functions and polymorphic programming can eliminate the need for **`switch`** logic





# 10.3 `virtual` Functions

- **`virtual`** functions
  - Suppose a set of shape classes such as **`Circle`**, **`Triangle`**, etc.
  - Every shape has own unique draw function but possible to call them by calling the **`draw`** function of base class **`Shape`**
    - Compiler determines dynamically (i.e., at run time) which to call
  - In base-class declare **`draw`** to be **`virtual`**
  - Override **`draw`** in each of the derived classes
  - **`virtual`** declaration:
    - Keyword **`virtual`** before function prototype in base-class

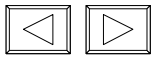
```
virtual void draw() const;
```

  - A base-class pointer to a derived class object will call the correct **`draw`** function

```
Shape->draw();
```

  - If a derived class does not define a **`virtual`** function, the function is inherited from the base class





# 10.3 Virtual Functions

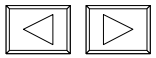
```
ShapePtr->Draw();
```

- Compiler implements dynamic binding
- Function determined during execution time

```
ShapeObject.Draw();
```

- Compiler implements static binding
- Function determined during compile-time



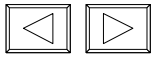


# 10.4 Abstract and Concrete Classes

- Abstract classes
  - Sole purpose is to provide a base class for other classes
  - No objects of an abstract base class can be instantiated
    - Too generic to define real objects (i.e., **`TwoDimensionalShape`**)
    - Can have pointers and references
- Concrete classes
  - Classes that can instantiate objects
  - Provide specifics to make real objects (i.e., **`Square`**, **`Circle`**)
- Making abstract classes
  - Declare one or more **`virtual`** functions as "pure" by initializing the function to zero
  - Example of a pure **`virtual`** function:

```
virtual double earnings() const = 0;
```



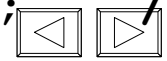


# 10.5 Polymorphism

- Polymorphism
  - Ability for objects of different classes to respond differently to the same function call
  - Implemented through **`virtual`** functions
    - Base-class pointer (or reference) calls a **`virtual`** function
    - C++ chooses the correct overridden function in object
  - If non-**`virtual`** member function defined in multiple classes and called from base-class pointer then the base-class version is used
    - If called from derived-class pointer then derived-class version is used
  - Suppose **`print`** is not a **`virtual`** function

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();    // call base-class print function
hPtr->print();    // call derived-class print
function
ePtr=&h;          // allowable implicit conversion
ePtr->print();    // still calls base-class print
```

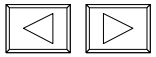


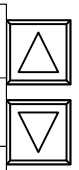


# 10.6 Case Study: A Payroll System Using Polymorphism

- The following example is a payroll system
  - Uses **`virtual`** functions and polymorphism to perform payroll calculations based on the type of an employee







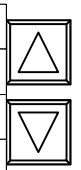
## Outline

**1. `Employee` Definition (base class)**

```
1  // Fig. 10.1: employ2.h
2  // Abstract base class Employee
3  #ifndef EMPLOY2_H
4  #define EMPLOY2_H
5
6  class Employee {
7  public:
8     Employee( const char *, const char * );
9     ~Employee();   // destructor reclaims memory
10    const char *getFirstName() const;
11    const char *getLastName() const;
12
13    // Pure virtual function makes Employee abstract base class
14    virtual double earnings() const = 0;  // pure virtual
15    virtual void print() const;            // virtual
16 private:
17    char *firstName;
18    char *lastName;
19 };
20
21 #endif
```

**`earnings`** is declared pure **`virtual`** because the implementation will depend on which derived class it will be used in.

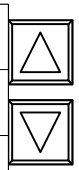
**`Employee`** is an abstract base class.



Outline

**1.1 Function Definitions**

```
22 // Fig. 10.1: employ2.cpp
23 // Member function definitions for
24 // abstract base class Employee.
25 // Note: No definitions given for pure virtual functions.
26 #include <iostream>
27
28 using std::cout;
29
30 #include <cstring>
31 #include <cassert>
32 #include "employ2.h"
33
34 // Constructor dynamically allocates space for the
35 // first and last name and uses strcpy to copy
36 // the first and last names into the object.
37 Employee::Employee( const char *first, const char *last )
38 {
39    firstName = new char[ strlen( first ) + 1 ];
40    assert( firstName != 0 );    // test that new worked
41    strcpy( firstName, first );
42
43    lastName = new char[ strlen( last ) + 1 ];
44    assert( lastName != 0 );     // test that new worked
45    strcpy( lastName, last );
46 }
47
48 // Destructor deallocates dynamically allocated memory
49 Employee::~Employee()
50 {
51    delete [] firstName;
52    delete [] lastName;
53 }
```

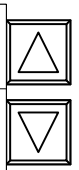


Outline

## 1.1 Function Definitions

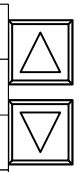
```
54
55 // Return a pointer to the first name
56 // Const return type prevents caller from modifying private
57 // data. Caller should copy returned string before destructor
58 // deletes dynamic storage to prevent undefined pointer.
59 const char *Employee::getFirstName() const
60 {
61    return firstName;   // caller must delete memory
62 }
63
64 // Return a pointer to the last name
65 // Const return type prevents caller from modifying private
66 // data. Caller should copy returned string before destructor
67 // deletes dynamic storage to prevent undefined pointer.
68 const char *Employee::getLastName() const
69 {
70    return lastName;   // caller must delete memory
71 }
72
73 // Print the name of the Employee
74 void Employee::print() const
75    { cout << firstName << ' ' << lastName; }
```

```
76 // Fig. 10.1: boss1.h
77 // Boss class derived from Employee
78 #ifndef BOSS1_H
79 #define BOSS1_H
80 #include "employ2.h"
81
82 class Boss : public Employee {
83 public:
84    Boss( const char *, const char *, double = 0.0 );
85    void setWeeklySalary( double );
86    virtual double earnings() const;
87    virtual void print() const;
88 private:
89    double weeklySalary;
90 };
91
92 #endif
```



## Outline

**1. `Boss` Definition (derived class)**



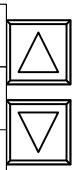
Outline

**1.1 Function Definitions**

```
93 // Fig. 10.1: boss1.cpp
94 // Member function definitions for class Boss
95 #include <iostream>
96
97 using std::cout;
98
99 #include "boss1.h"
100
101// Constructor function for class Boss
102Boss::Boss( const char *first, const char *last, double s )
103   : Employee( first, last )  // call base-class constructor
104{ setWeeklySalary( s ); }
105
106// Set the Boss's salary
107void Boss::setWeeklySalary( double s )
108   { weeklySalary = s > 0 ? s : 0; }
109
110// Get the Boss's pay
111double Boss::earnings() const { return weeklySalary; }
112
113// Print the Boss's name
114void Boss::print() const
115{
116   cout << "\n             Boss: ";
117   Employee::print();
118}
```

Notice the overriden **`earnings`** and **`print`** functions.

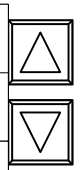
They were declared **`virtual`** in the base class.



## Outline

**1. `CommissionWorker` Definition (derived class)**

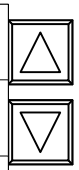
```
119// Fig. 10.1: commis1.h
120// CommissionWorker class derived from Employee
121#ifndef COMMIS1_H
122#define COMMIS1_H
123#include "employ2.h"
124
125class CommissionWorker : public Employee {
126public:
127   CommissionWorker( const char *, const char *,
128                     double = 0.0, double = 0.0,
129                     int = 0 );
130   void setSalary( double );
131   void setCommission( double );
132   void setQuantity( int );
133   virtual double earnings() const;
134   virtual void print() const;
135private:
136   double salary;        // base salary per week
137   double commission;    // amount per item sold
138   int quantity;         // total items sold for week
139};
140
141#endif
```



Outline

**1.1 Function Definitions**

```
142// Fig. 10.1: commis1.cpp
143// Member function definitions for class CommissionWorker
144#include <iostream>
145
146using std::cout;
147
148#include "commis1.h"
149
150// Constructor for class CommissionWorker
151CommissionWorker::CommissionWorker( const char *first,
152         const char *last, double s, double c, int q )
153   : Employee( first, last )  // call base-class constructor
154{
155   setSalary( s );
156   setCommission( c );
157   setQuantity( q );
158}
159
160// Set CommissionWorker's weekly base salary
161void CommissionWorker::setSalary( double s )
162   { salary = s > 0 ? s : 0; }
```



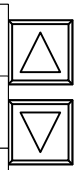
Outline

**1.1 Function Definitions**

```
163
164// Set CommissionWorker's commission
165void CommissionWorker::setCommission( double c )
166   { commission = c > 0 ? c : 0; }
167
168// Set CommissionWorker's quantity sold
169void CommissionWorker::setQuantity( int q )
170   { quantity = q > 0 ? q : 0; }
171
172// Determine CommissionWorker's earnings
173double CommissionWorker::earnings() const
174   { return salary + commission * quantity; }
175
176// Print the CommissionWorker's name
177void CommissionWorker::print() const
178{
179   cout << "\nCommission worker: ";
180   Employee::print();
181}
```

Notice the overriden **`earnings`** and **`print`** functions.

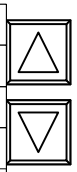
They were declared **`virtual`** in the base class.



Outline

**1. `PieceWorker` Definition (derived class)**

```
182// Fig. 10.1: piece1.h
183// PieceWorker class derived from Employee
184#ifndef PIECE1_H
185#define PIECE1_H
186#include "employ2.h"
187
188class PieceWorker : public Employee {
189public:
190   PieceWorker( const char *, const char *,
191                double = 0.0, int = 0);
192   void setWage( double );
193   void setQuantity( int );
194   virtual double earnings() const;
195   virtual void print() const;
196private:
197   double wagePerPiece; // wage for each piece output
198   int quantity;        // output for week
199};
200
201#endif
```



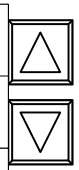
Outline

## 1.1 Function Definitions

```
202// Fig. 10.1: piece1.cpp
203// Member function definitions for class PieceWorker
204#include <iostream>
205
206using std::cout;
207
208#include "piece1.h"
209
210// Constructor for class PieceWorker
211PieceWorker::PieceWorker( const char *first, const char *last,
212                          double w, int q )
213   : Employee( first, last )  // call base-class constructor
214{
215   setWage( w );
216   setQuantity( q );
217}
218
219// Set the wage
220void PieceWorker::setWage( double w )
221   { wagePerPiece = w > 0 ? w : 0; }
222
223// Set the number of items output
224void PieceWorker::setQuantity( int q )
225   { quantity = q > 0 ? q : 0; }
226
227// Determine the PieceWorker's earnings
228double PieceWorker::earnings() const
229   { return quantity * wagePerPiece; }
230
231// Print the PieceWorker's name
232void PieceWorker::print() const
233{
234   cout << "\n     Piece worker: ";
235   Employee::print();
236}
```

Again, notice the overridden **`earnings`** and **`print`** functions.

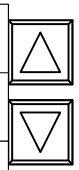
They were declared **`virtual`** in the base class.



Outline

**1. `HourlyWorker` Definition (derived class)**

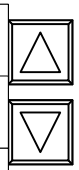
```
237// Fig. 10.1: hourly1.h
238// Definition of class HourlyWorker
239#ifndef HOURLY1_H
240#define HOURLY1_H
241#include "employ2.h"
242
243class HourlyWorker : public Employee {
244public:
245   HourlyWorker( const char *, const char *,
246                 double = 0.0, double = 0.0);
247   void setWage( double );
248   void setHours( double );
249   virtual double earnings() const;
250   virtual void print() const;
251private:
252   double wage;   // wage per hour
253   double hours;  // hours worked for week
254};
255
256#endif
```



Outline

## 1.1 Function Definitions

```
257// Fig. 10.1: hourly1.cpp
258// Member function definitions for class HourlyWorker
259#include <iostream>
260
261using std::cout;
262
263#include "hourly1.h"
264
265// Constructor for class HourlyWorker
266HourlyWorker::HourlyWorker( const char *first,
267                            const char *last,
268                            double w, double h )
269   : Employee( first, last )   // call base-class constructor
270{
271   setWage( w );
272   setHours( h );
273}
274
275// Set the wage
276void HourlyWorker::setWage( double w )
277   { wage = w > 0 ? w : 0; }
```

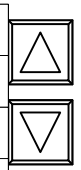


Outline

**1.1 Function Definitions**

```
278
279// Set the hours worked
280void HourlyWorker::setHours( double h )
281   { hours = h >= 0 && h < 168 ? h : 0; }
282
283// Get the HourlyWorker's pay
284double HourlyWorker::earnings() const
285{
286   if ( hours <= 40 ) // no overtime
287      return wage * hours;
288   else                // overtime is paid at wage * 1.5
289      return 40 * wage + ( hours - 40 ) * wage * 1.5;
290}
291
292// Print the HourlyWorker's name
293void HourlyWorker::print() const
294{
295   cout << "\n    Hourly worker: ";
296   Employee::print();
297}
```

Overridden functions.



Outline

**1. Load headers**

```
298// Fig. 10.1: fig10_01.cpp
299// Driver for Employee hierarchy
300#include <iostream>
301
302using std::cout;
303using std::endl;
304
305#include <iomanip>
306
307using std::ios;
308using std::setiosflags;
309using std::setprecision;
310
311#include "employ2.h"
312#include "boss1.h"
313#include "commis1.h"
314#include "piece1.h"
315#include "hourly1.h"
316
317void virtualViaPointer( const Employee * );
318void virtualViaReference( const Employee & );
319
320int main()
321{
322   // set output formatting
323   cout << setiosflags( ios::fixed | ios::showpoint )
324        << setprecision( 2 );
325
```

## Outline

**1.1 Initialize objects**

**2. Print**

```
326   Boss b( "John", "Smith", 800.00 );
327   b.print();
328   cout << " earned $" << b.earnings();     // static binding
329   virtualViaPointer( &b );                 // uses dynamic binding
330   virtualViaReference( b );                // uses dynamic binding
331
332   CommissionWorker c( "Sue", "Jones", 200
333   c.print();                               // static binding
334   cout << " earned $" << c.earnings();
335   virtualViaPointer( &c );                 // u
336   virtualViaReference( c );                // u
337
338   PieceWorker p( "Bob", "Lewis", 2.5, 200
339   p.print();
340   cout << " earned $" << p.earnings();
341   virtualViaPointer( &p );                 // u
342   virtualViaReference( p );                // u
343
344   HourlyWorker h( "Karen", "Price", 13.75
345   h.print();
346   cout << " earned $" << h.earnings();     // static binding
347   virtualViaPointer( &h );                 // uses dynamic binding
348   virtualViaReference( h );
349   cout << endl;
350   return 0;
351}
352
353// Make virtual function calls off a base-class pointer
354// using dynamic binding.
355void virtualViaPointer( const Employee *baseClassPtr )
356{
357   baseClassPtr->print();
358   cout << " earned $" << baseClassPtr->earnings();
359}
```

```
Boss: John Smith earned $800.00
```

```
Boss: John Smith earned $800.00
```

```
Boss: John Smith earned $800.00
```

Can function **print** using the object itself.

```
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
```

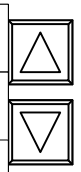
dynamic binding

```
Piece worker: Bob Lewis earned $500.00
    Piece worker: Bob Lewis earned $500.00
    Piece worker: Bob Lewis earned $500.00
```

This uses **virtual** functions and dynamic binding.

```
Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
```

Take in a baseclass pointer, call the **virtual** function **print**.



Outline

### 3. Function Definitions

```
360
361// Make virtual function calls off a base-class reference
362// using dynamic binding.
363void virtualViaReference( const Employee &baseClassRef )
364{
365   baseClassRef.print();
366   cout << " earned $" << baseClassRef.earnings();
367}
```

Take in base class reference, call the **`virtual`** function **`print`**.

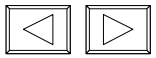
### Program Output

```
Boss: John Smith earned $800.00
             Boss: John Smith earned $800.00
             Boss: John Smith earned $800.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
```

# 10.7 New Classes and Dynamic Binding

- Polymorphism and virtual functions
  - Work well when all classes are not known in advance
  - Use dynamic binding to accommodate new classes being added to a system
- Dynamic binding (late binding)
  - Object's type need not be know at compile time for a **`virtual`** function
  - **`virtual`** function call is matched at run time

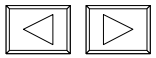




# 10.8 Virtual Destructors

- Problem:
  - If a base-class pointer to a derived object is deleted, the base-class destructor will act on the object
- Solution:
  - declare a **`virtual`** base-class destructor to ensure that the appropriate destructor will be called

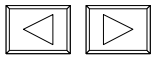


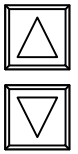


# 10.9 Case Study: Inheriting Interface and Implementation

- Extension of point, circle, cylinder hierarchy
  - Use the abstract base class **`Shape`** to head the hierarchy
    - Two pure virtual functions **`printShapeName`** and **`print`**
    - Two other virtual functions **`volume`** and **`area`**
  - **`Point`** is derived from **`Shape`** and inherits these implementations







## Outline

**1. `Shape` Definition (abstract base `class`)**

---

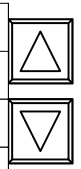
**1. `Point` Definition (derived `class`)**

```
1  // Fig. 10.2: shape.h
2  // Definition of abstract base class Shape
3  #ifndef SHAPE H
4  #define SHAPE H
5
6  class Shape {
7  public:
8     virtual double area() const { return 0.0; }
9     virtual double volume() const { return 0.0; }
10
11    // pure virtual functions overridden in derived classes
12    virtual void printShapeName() const = 0;
13    virtual void print() const = 0;
14 };
15
16 #endif
17 // Fig. 10.2: point1.h
18 // Definition of class Point
19 #ifndef POINT1 H
20 #define POINT1 H
21
22 #include <iostream>
23
24 using std::cout;
25
26 #include "shape.h"
27
28 class Point : public Shape {
29 public:
30    Point( int = 0, int = 0 );  // default constructor
31    void setPoint( int, int );
32    int getX() const { return x; }
33    int getY() const { return y; }
```

Notice the **`virtual`** functions which will be overridden by each class.

**`Point`** inherits from the abstract base class.

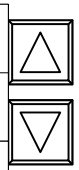
```
34    virtual void printShapeName() const { cout << "Point: "; }
35    virtual void print() const;
36 private:
37    int x, y;   // x and y coordinates of Point
38 };
39
40 #endif
41 // Fig. 10.2: point1.cpp
42 // Member function definitions for class Point
43 #include "point1.h"
44
45 Point::Point( int a, int b ) { setPoint( a, b ); }
46
47 void Point::setPoint( int a, int b )
48 {
49    x = a;
50    y = b;
51 }
52
53 void Point::print() const
54    { cout << '[' << x << ", " << y << ']'; }
```



## Outline

**1. `Point` Definition (derived `class`)**

**1.1 Function Definitions**

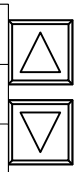


Outline

**1. Circle Definition (derived class)**

```
55 // Fig. 10.2: circle1.h
56 // Definition of class Circle
57 #ifndef CIRCLE1_H
58 #define CIRCLE1_H
59 #include "point1.h"
60
61 class Circle : public Point {
62 public:
63    // default constructor
64    Circle( double r = 0.0, int x = 0, int y = 0 );
65
66    void setRadius( double );
67    double getRadius() const;
68    virtual double area() const;
69    virtual void printShapeName() const { cout << "Circle: "; }
70    virtual void print() const;
71 private:
72    double radius;   // radius of Circle
73 };
74
75 #endif
```

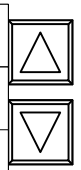
**`Circle`** inherits from **`Point`**.



Outline

**1.1 Function Definitions**

```
76 // Fig. 10.2: circle1.cpp
77 // Member function definitions for class Circle
78 #include <iostream>
79
80 using std::cout;
81
82 #include "circle1.h"
83
84 Circle::Circle( double r, int a, int b )
85    : Point( a, b )  // call base-class constructor
86 { setRadius( r ); }
87
88 void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }
89
90 double Circle::getRadius() const { return radius; }
91
92 double Circle::area() const
93    { return 3.14159 * radius * radius; }
94
95 void Circle::print() const
96 {
97    Point::print();
98    cout << "; Radius = " << radius;
99 }
```

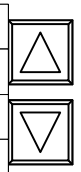


Outline

**1. `Cylinder` Definition (derived `class`)**

```
100 // Fig. 10.2: cylindr1.h
101 // Definition of class Cylinder
102 #ifndef CYLINDR1_H
103 #define CYLINDR1_H
104 #include "circle1.h"
105
106 class Cylinder : public Circle {
107 public:
108    // default constructor
109    Cylinder( double h = 0.0, double r = 0.0,
110              int x = 0, int y = 0 );
111
112    void setHeight( double );
113    double getHeight();
114    virtual double area() const;
115    virtual double volume() const;
116    virtual void printShapeName() const { cout << "Cylinder: "; }
117    virtual void print() const;
118 private:
119    double height;   // height of Cylinder
120 };
121
122 #endif
```

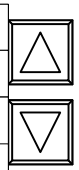
**`Cylinder`** inherits from **`Circle`**.



Outline

## 1.1 Function Definitions

```
123// Fig. 10.2: cylindr1.cpp
124// Member and friend function definitions for class Cylinder
125#include <iostream>
126
127using std::cout;
128
129#include "cylindr1.h"
130
131Cylinder::Cylinder( double h, double r, int x, int y )
132   : Circle( r, x, y )  // call base-class constructor
133{ setHeight( h ); }
134
135void Cylinder::setHeight( double h )
136   { height = h > 0 ? h : 0; }
137
138double Cylinder::getHeight() { return height; }
139
140double Cylinder::area() const
141{
142   // surface area of Cylinder
143   return 2 * Circle::area() +
144          2 * 3.14159 * getRadius() * height;
145}
146
147double Cylinder::volume() const
148   { return Circle::area() * height; }
149
150void Cylinder::print() const
151{
152   Circle::print();
153   cout << "; Height = " << height;
154}
```



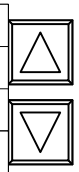
Outline

1. Load headers

1.1 Function prototypes

1.2 Initialize objects

```
155// Fig. 10.2: fig10_02.cpp
156// Driver for shape, point, circle, cylinder hierarchy
157#include <iostream>
158
159using std::cout;
160using std::endl;
161
162#include <iomanip>
163
164using std::ios;
165using std::setiosflags;
166using std::setprecision;
167
168#include "shape.h"
169#include "point1.h"
170#include "circle1.h"
171#include "cylindr1.h"
172
173void virtualViaPointer( const Shape * );
174void virtualViaReference( const Shape & );
175
176int main()
177{
178   cout << setiosflags( ios::fixed | ios::showpoint )
179        << setprecision( 2 );
180
181   Point point( 7, 11 );                  // create a Point
182   Circle circle( 3.5, 22, 8 );           // create a Circle
183   Cylinder cylinder( 10, 3.3, 10, 10 );  // create a Cylinder
184
185   point.printShapeName();    // static binding
```



Outline

**2. Function calls**

```
186   point.print();                // static binding
187   cout << '\n';
188
189   circle.printShapeName();      // static binding
190   circle.print();
191   cout << '\n';
192
193   cylinder.printShapeName(); // static binding
194   cylinder.print();
195   cout << "\n\n";
196
197   Shape *arrayOfShapes[ 3 ];  // array of base-class pointers
198
199   // aim arrayOfShapes[0] at derived-class Point object
200   arrayOfShapes[ 0 ] = &point;
201
202   // aim arrayOfShapes[1] at derived-class Ci
203   arrayOfShapes[ 1 ] = &circle;
204
205   // aim arrayOfShapes[2] at derived-class Cy
206   arrayOfShapes[ 2 ] = &cylinder;
207
208   // Loop through arrayOfShapes and c
209   // to print the shape name, attribu
210   // of each object using dynamic binding.
211   cout << "Virtual function calls made off "
212        << "base-class pointers\n";
213
214   for ( int i = 0; i < 3; i++ )
215      virtualViaPointer( arrayOfShapes[ i ] );
216
217   // Loop through arrayOfShapes and call virtua
218   // to print the shape name, attributes, area, and volume
219   // of each object using dynamic binding.
```

```
Point: [7, 11]
```

```
Print
```

```
Circle: [22, 8]; Radius = 3.50
```

```
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
```

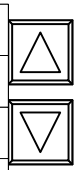
Create an array of base class pointers. Assign these to the objects, then call the **`print`** functions again, using the base class pointers. The appropriate **`virtual`** functions will be called

```
Virtual function calls made off base-class pointers
```

```
Point: [7, 11]
Area
Volu
```

```
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
```

```
ght =
10.00
Area = 275.77
Volume = 342.12
```



Outline

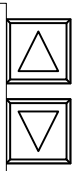
**2. Function calls**

**Definitions**

```
220   cout << "Virtual function calls made off "
221        << "base-class references\n";
222
223   for ( int j = 0; j < 3; j++ )
224      virtualViaReference( *arrayOfShapes[ j ] );
225
226   return 0;
227}
228
229// Make virtual function calls
230// using dynamic binding.
231void virtualViaPointer( const S
232{
233   baseClassPtr->printShapeName
234   baseClassPtr->print();
235   cout << "\nArea = " << baseC
236        << "\nVolume = " << bas
237}
238
239// Make virtual function calls off a base-class reference
240// using dynamic binding.
241void virtualViaReference( const Shape &baseClassRef )
242{
243   baseClassRef.printShapeName();
244   baseClassRef.print();
245   cout << "\nArea = " << baseClassRef.area()
246        << "\nVolume = " << baseClassRef.volume() << "\n\n";
247}
```

Repeat process using base-class

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

Point: [7, 11]

Area = 0.00

Volume = 0.00

Circle: [22, 8]; Radius = 3.50

Area = 38.48

Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height =
10.00

Area = 275.77

Volume = 342.12
```



Outline

**Program Output**

```
Point: [7, 11]
Circle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Virtual function calls made off base-class pointers
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
Virtual function calls made off base-class references
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
```

# 10.10 Polymorphism, `virtual` Functions and Dynamic Binding "Under the Hood"

- When to use polymorphism
  - Polymorphism requires a lot of overhead
  - Polymorphism is 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, its vtable has a pointer to the proper function
    - If a derived class has the same function as a base class, then the function pointer points to the base-class function
  - Detailed explanation in Fig. 10.3