



Πανεπιστήμιο Πατρών

Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών και
Πληροφορικής

ΟΝΤΟΚΕΝΤΡΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ (C++)

Πολυμορφισμός

Περιεχόμενα

- **Εισαγωγή**
- **Σχέσεις αντικειμένων σε μια ιεραρχία κληρονομικότητας**
 - Κλήση μεθόδων της κλάσης βάσης από αντικείμενα της παραγόμενης κλάσης
 - Δείκτης παραγόμενης κλάσης σε αντικείμενο της κλάσης βάσης
 - Κλήση μεθόδων παραγόμενης κλάσης μέσω δείκτη κλάσης βάσης
 - Εικονικές Συναρτήσεις (Virtual Functions)
- **Παραδείγματα Πολυμορφισμού**
- **Χρήση πεδίου για τύπο αντικειμένων και εντολής switch**
- **Abstract Classes**
- **Μελέτη Περίπτωσης : Κληρονόμηση Διεπαφής και Υλοποίησης**
- **Πολυμορφισμός, Εικονικές Συναρτήσεις και Δυναμική Σύνδεση: Υπάρχει κόστος χρήσης?**
- **Μελέτη Περίπτωσης: Σύστημα Μισθοδοσίας με Πολυμορφισμό**

Εισαγωγή

- Πολυμορφισμός
 - Απαιτείται η ύπαρξη μιας ιεραρχίας κλάσεων
 - Χρησιμοποιούμε αντικείμενα που ανήκουν στην ίδια ιεραρχία κλάσεων ως να ήταν όλα αντικείμενα της κλάσης βάσης
 - Εικονικές συναρτήσεις + δυναμική σύνδεση (Virtual functions + dynamic binding)
 - Τεχνολογίες που υλοποιούν τον πολυμορφισμό
 - Επιτρέπει την αποτελεσματική επέκταση των προγραμμάτων
 - Νέες κλάσεις μπορούν εύκολα να προστεθούν και να υποστούν επεξεργασία όπως και οι υπάρχουσες
 - Binding: σύνδεση της κλήσης μιας μεθόδου με το σώμα της μεθόδου που τελικά θα κληθεί
 - Early (ή static) binding: καθορίζεται κατά το compile time
 - Late (ή dynamic) binding: καθορίζεται κατά το run time

Ένα πρώτο παράδειγμα

- Κλάση βάσης: *Animal*
- Παραγόμενες κλάσεις: *Fish, Frog, Bird*
- Η μέθοδος *move()* είναι κοινή, αλλά υλοποιείται διαφορετικά για κάθε είδος *Animal*.
- Μια εφαρμογή κρατά έναν πίνακα από αντικείμενα των παραγόμενων κλάσεων και κάθε 1sec στέλνει ένα μήνυμα *move* σε κάθε αντικείμενο.
- Κάθε αντικείμενο τύπου *Animal* που δέχεται το μήνυμα γνωρίζει πως θα πρέπει να αντιδράσει.
- Το ίδιο μήνυμα που στέλνεται σε διαφορετικά αντικείμενα (του ίδιου βασικού τύπου *Animal*) έχει πολλές μορφές αποτελεσμάτων → πολυμορφισμός
- Πόσο εύκολο είναι να προσθέσουμε μια νέα κλάση *Turtle*?

Σχέσεις αντικειμένων σε μια ιεραρχία κληρονομικότητας

- Σημείο κλειδί
 - Ένα αντικείμενο μιας παραγόμενης κλάσης μπορεί να το μεταχειριστούμε και ως αντικείμενο της κλάσης βάσης
 - “IS-A” σχέση
 - Όμως, το αντίστροφο δεν ισχύει: ένα αντικείμενο της κλάσης βάσης δεν είναι αντικείμενο της παραγόμενης κλάσης
- Εκχώρηση αντικειμένου παραγόμενης κλάσης σε αντικείμενο τύπου κλάσης βάσης (με τιμή)
 - Object Slicing!

```
class Pet
{
public:
    string name;
};
class Dog : public Pet
{
public:
    string breed;
};

int main()
{
    Dog dog;
    Pet pet;

    dog.name = "Tommy";
    dog.breed = "Kangal Dog";
    pet = dog;
    cout << pet.breed; //ERROR
```

Κλήση μεθόδων της κλάσης βάσης από αντικείμενα της παραγόμενης κλάσης

- Τρεις διαφορετικοί τρόποι μεταχείρισης δεικτών σε αντικείμενα μιας ιεραρχίας κλάσεων
 - Δείκτης κλάσης βάσης σε αντικείμενο της κλάσης βάσης
 - Δείκτης παραγόμενης κλάσης σε αντικείμενο της παραγόμενης κλάσης
 - Δείκτης κλάσης βάσης σε αντικείμενο της παραγόμενης κλάσης
 - Εφικτό εξ' αιτίας της “IS-A” σχέσης (**Circle** “is a” **Point**)
 - Επιφέρει την κλήση μεθόδων της κλάσης βάσης
 - “Upcasting” (γίνεται implicitly)
 - Η κλήση συναρτήσεων εξαρτάται από την κλάση του δείκτη/handle
 - Δεν εξαρτάται από το αντικείμενο στο οποίο δείχνει ο δείκτης
 - Αυτό μπορεί να αλλάξει με τη χρήση των **virtual** functions



```

1 // Fig. 10.1: point.h
2 // Point class definition represents an x-y coordinate pair.
3 #ifndef POINT_H
4 #define POINT_H
5
6 class Point {
7
8 public:
9     Point( int = 0, int = 0 ); // default constructor
10
11     void setX( int ); // set x in coordinate pair
12     int getX() const; // return x from coordinate pair
13
14     void setY( int ); // set y in coordinate pair
15     int getY() const; // return y from coordinate pair
16
17     void print() const; // output Point object
18
19 private:
20     int x; // x part of coordinate pair
21     int y; // y part of coordinate pair
22
23 }; // end class Point
24
25 #endif

```

Base class print function.



```

1 // Fig. 10.2: point.cpp
2 // Point class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "point.h" // Point class definition
8
9 // default constructor
10 Point::Point( int xValue, int yValue )
11     : x( xValue ), y( yValue )
12 {
13     // empty body
14 } // end Point constructor
15
16 // set x in coordinate pair
17 void Point::setX( int xValue )
18 {
19     x = xValue; // no need for validation
20 } // end function setX
21
22
23

```



```

24 // return x from coordinate pair
25 int Point::getX() const
26 {
27     return x;
28 }
29 // end function getX
30
31 // set y in coordinate pair
32 void Point::setY( int yValue )
33 {
34     y = yValue; // no need for validation
35 }
36 // end function setY
37
38 // return y from coordinate pair
39 int Point::getY() const
40 {
41     return y;
42 }
43 // end function getY
44
45 // output Point object
46 void Point::print() const
47 {
48     cout << '[' << getX() << ", " << getY() << '>';
49 }
50 // end function print

```

Output the x,y coordinates of the Point.



```

1 // Fig. 10.3: circle.h
2 // Circle class contains x-y coordinate pair and radius.
3 #ifndef CIRCLE_H
4 #define CIRCLE_H
5
6 #include "point.h" // Point class definition
7
8 class Circle : public Point {
9
10 public:
11
12     // default constructor
13     Circle( int = 0, int = 0, double =
14
15     void setRadius( double ); // set radius
16     double getRadius() const; // return radius
17
18     double getDiameter() const; // return diameter
19     double getCircumference() const; // return circumference
20     double getArea() const; // return area
21
22     void print() const; // output Circle object
23
24 private:
25     double radius; // Circle's radius
26
27 }; // end class Circle
28
29 #endif

```

Circle inherits from Point, but redefines its print function.

```

1 // Fig. 10.4: circle.cpp
2 // Circle class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "circle.h" // Circle class definition
8
9 // default constructor
10 Circle::Circle( int xValue, int yValue, double radiusValue )
11     : Point( xValue, yValue ) // call base-class constructor
12 {
13     setRadius( radiusValue );
14
15 } // end Circle constructor
16
17 // set radius
18 void Circle::setRadius( double radiusValue )
19 {
20     radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
21
22 } // end function setRadius
23

```

```

24 // return radius
25 double Circle::getRadius() const
26 {
27     return radius;
28
29 } // end function getRadius
30
31 // calculate and return diameter
32 double Circle::getDiameter() const
33 {
34     return 2 * getRadius();
35
36 } // end function getDiameter
37
38 // calculate and return circumference
39 double Circle::getCircumference() const
40 {
41     return 3.14159 * getDiameter();
42
43 } // end function getCircumference
44
45 // calculate and return area
46 double Circle::getArea() const
47 {
48     return 3.14159 * getRadius() * getRadius();
49
50 } // end function getArea

```

```

51
52 // output Circle object
53 void Circle::print() const
54 {
55     cout << "center = ";
56     Point::print(); // invoke Point's print f
57     cout << "; radius = " << getRadius();
58
59 } // 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.

```

1 // Fig. 10.5: fig10_05.cpp
2 // Aiming base-class and derived-class pointers at base-class
3 // and derived-class objects, respectively.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11
12 using std::setprecision;
13
14 #include "point.h" // Point class definition
15 #include "circle.h" // Circle class definition
16
17 int main()
18 {
19     Point point( 30, 50 );
20     Point *pointPtr = 0; // base-class pointer
21
22     Circle circle( 120, 89, 2.7 );
23     Circle *circlePtr = 0; // derived-class pointer
24

```



Outline

fig10_05.cpp
(2 of 3)

```

25 // set floating-point numeric format
26 cout << fixed << setprecision( 2 );
27
28 // output objects point and circle
29 cout << "Print point and circle objects\n";
30     << "\nPoint: ";
31 point.print(); // invokes Point's print
32 cout << "\nCircle: ";
33 circle.print(); // invokes Circle's print
34
35 // aim base-class pointer at base-class object and print
36 pointPtr = &point;
37 cout << "\n\nCalling print with base-class pointer to "
38     << "\nbase-class object invokes base-class print "
39     << "function:\n";
40 pointPtr->print(); // invokes Point's print
41
42 // aim derived-class pointer at derived-class object
43 // and print
44 circlePtr = &circle;
45 cout << "\n\nCalling print with derived-class pointer to "
46     << "\nderived-class object invokes derived-class "
47     << "print function:\n";
48 circlePtr->print(); // invokes Circle's print
49

```

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.

© 2003 Prentice Hall, Inc.
All rights reserved.



Outline

fig10_05.cpp
(3 of 3)

```

50 // aim base-class pointer at derived-class object and print
51 pointPtr = &circle;
52 cout << "\n\nCalling print with base-class pointer to "
53     << "derived-class object\ninvokes base-class print "
54     << "function on that derived-class object:\n";
55 pointPtr->print(); // invokes Point's print
56 cout << endl;
57
58 return 0;
59
60 } // 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.

© 2003 Prentice Hall, Inc.
All rights reserved.



```

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]

```

Δείκτης παραγόμενης κλάσης σε αντικείμενο της κλάσης βάσης

- Στο προηγούμενο παράδειγμα είδαμε
 - Δείκτης κλάσης βάσης σε αντικείμενο της παραγόμενης κλάσης
 - **Circle** “is a” **Point**
- Δείκτης παραγόμενης κλάσης σε αντικείμενο της κλάσης βάσης
 - Compiler error
 - **Point** is not a **Circle**
 - Η κλάση **Circle** διαθέτει ιδιότητες/λειτουργίες που δεν διαθέτει η κλάση **Point**
 - Π.χ. **setRadius**
 - Μπορούμε να κάνουμε cast τη διεύθυνση του αντικειμένου της κλάσης βάσης σε δείκτη παραγόμενης κλάσης
 - Ονομάζεται downcasting
 - Επιτρέπει την προσπέλαση λειτουργικότητας της παραγόμενης κλάσης
 - Δεν είναι πάντα safe
 - Μπορεί να γίνει με `dynamic_cast<T>()`

```

1 // Fig. 10.6: fig10_06.cpp
2 // Aiming a derived-class pointer at a base-class object.
3 #include "point.h" // Point class definition
4 #include "circle.h" // Circle class definition
5
6 int main()
7 {
8     Point point( 30, 50 );
9     Circle *circlePtr = 0;
10
11     // aim derived-class pointer at base-class object
12     circlePtr = &point; // Error: a Point is not a Circle
13     // could be: circlePtr = (Circle *)&point;
14     return 0;
15
16 } // end main

```

```

C:\cpp4\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

```



Outline

fig10_06.cpp
(1 of 1)

fig10_06.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Κλήση μεθόδων παραγόμενης κλάσης μέσω δείκτη κλάσης βάσης

- Χρήση δείκτη/αναφοράς
 - Δείκτης κλάσης βάσης μπορεί να δείχνει σε αντικείμενο της παραγόμενης κλάσης
 - Μπορεί όμως να καλέσει μόνο τις μεθόδους της κλάσης βάσης
 - Η κλήση μεθόδων της παραγόμενης κλάσης συνιστά λάθος
 - Μέθοδοι που δεν ορίζονται στη κλάση βάσης
- Ο τύπος δεδομένων ενός/μιας δείκτη/αναφοράς προσδιορίζουν τι μεθόδους μπορούμε να καλέσουμε


fig10_07.cpp
 (1 of 2)

```

1 // Fig. 10.7: fig10_07.cpp
2 // Attempting to invoke derived-class-only member functions
3 // through a base-class pointer.
4 #include "point.h" // Point class definition
5 #include "circle.h" // Circle class definition
6
7 int main()
8 {
9     Point *pointPtr = 0;
10    Circle circle( 120, 89, 2.7 );
11
12    // aim base-class pointer at derived-class object
13    pointPtr = &circle;
14
15    // invoke base-class member functions on derived-class
16    // object through base-class pointer
17    int x = pointPtr->getX();
18    int y = pointPtr->getY();
19    pointPtr->setX( 10 );
20    pointPtr->setY( 10 );
21    pointPtr->print();
22

```


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**.



```

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'

```

Εικονικές Συναρτήσεις (Virtual Functions)

- Ο κανόνας είναι ο τύπος του δείκτη να καθορίζει τι μέθοδοι μπορούν να κληθούν
- Οι **virtual** functions μπορούν να τον αλλάξουν
 - το αντικείμενο (και όχι ο δείκτης) να καθορίζει τι μέθοδοι μπορούν να κληθούν
- Είναι αυτό χρήσιμο?
 - Ας θεωρήσουμε την ιεραρχία **Circle, Triangle, Rectangle → Shape**
 - Κάθε κλάση έχει τη δικιά της συνάρτηση **draw**
 - Για να σχεδιάσω οποιοδήποτε σχήμα
 - Καλούμε τη **draw** μέσω δείκτη της κλάσης βάσης **Shape**
 - Το πρόγραμμα επιλέγει την κατάλληλη συνάρτηση **draw** κατά το χρόνο εκτέλεσης (δυναμικά)
 - Μπορούμε να χειριστούμε με γενικό τρόπο όλα τα σχήματα

Εικονικές Συναρτήσεις (Virtual Functions)

- Δηλώνουμε την **draw** ως **virtual** στην κλάση βάσης
 - Υπερκαλύπτουμε τη **draw** σε κάθε παραγόμενη κλάση
 - σαν να την ξαναορίζουμε, αλλά η νέα συνάρτηση πρέπει να έχει ακριβώς την ίδια υπογραφή με αυτή της κλάσης βάσης
 - Εάν δηλώσουμε μια συνάρτηση **virtual** στην κλάση βάσης
 - **virtual void draw() const;**
 - είναι **virtual** σε όλες τις παραγόμενες κλάσεις
 - Είναι καλή πρακτική να δηλώνουμε ρητά ως **virtual** τις συναρτήσεις και στις παραγόμενες κλάσεις (program clarity)
 - C++ 11: προσδιοριστικό **override** (επίσης: **final**)
- Δυναμική σύνδεση (Dynamic binding)
 - Επιλογή κλήσης της κατάλληλης συνάρτησης κατά το χρόνο εκτέλεσης
 - Συμβαίνει μόνο με τη χρήση δεικτών/αναφορών (Γιατί;)
 - Εάν η συνάρτηση κληθεί με χρήση ονόματος αντικειμένου (π.χ. **squareObject.draw()**), τότε χρησιμοποιείται η συνάρτηση του αντικειμένου (static binding)
 - Object Slicing

```

1 // Fig. 10.8: point.h
2 // Point class definition represents an x-y coordinate pair.
3 #ifndef POINT_H
4 #define POINT_H
5
6 class Point {
7
8 public:
9     Point( int = 0, int = 0 ); // default constructor
10
11     void setX( int ); // set x in coordinate pair
12     int getX() const; // return x from coordinate pair
13
14     void setY( int ); // set y in
15     int getY() const; // return y
16
17     virtual void print() const; // output Point object
18
19 private:
20     int x; // x part of coordinate pair
21     int y; // y part of coordinate pair
22
23 }; // end class Point
24
25 #endif

```

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



Outline

26

point.h (1 of 1)



```

1 // Fig. 10.9: circle.h
2 // Circle class contains x-y coordinate pair and radius.
3 #ifndef CIRCLE_H
4 #define CIRCLE_H
5
6 #include "point.h" // Point class definition
7
8 class Circle : public Point {
9
10 public:
11
12     // default constructor
13     Circle( int = 0, int = 0, double = 0.0 );
14
15     void setRadius( double ); // set radius
16     double getRadius() const; // return radius
17
18     double getDiameter() const; // return diameter
19     double getCircumference() const; // return circumference
20     double getArea() const; // return area
21
22     virtual void print() const; // output Circle object
23
24 private:
25     double radius; // Circle's radius
26
27 }; // end class Circle
28
29 #endif

```

© 2003 Prentice Hall, Inc.
All rights reserved.



```

1 // Fig. 10.10: fig10_10.cpp
2 // Introducing polymorphism, virtual functions and dynamic
3 // binding.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11
12 using std::setprecision;
13
14 #include "point.h" // Point class definition
15 #include "circle.h" // Circle class definition
16
17 int main()
18 {
19     Point point( 30, 50 );
20     Point *pointPtr = 0;
21
22     Circle circle( 120, 89, 2.7 );
23     Circle *circlePtr = 0;
24

```

© 2003 Prentice Hall, Inc.
All rights reserved.



```

25 // set floating-point numeric formatting
26 cout << fixed << setprecision( 2 );
27
28 // output objects point and circle using static binding
29 cout << "Invoking print function on point and circle "
30     << "\nobjects with static binding "
31     << "\n\nPoint: ";
32 point.print(); // static binding
33 cout << "\nCircle: ";
34 circle.print(); // static binding
35
36 // output objects point and circle using dynamic binding
37 cout << "\n\nInvoking print function on point and circle "
38     << "\nobjects with dynamic binding";
39
40 // aim base-class pointer at base-class object and print
41 pointPtr = &point;
42 cout << "\n\nCalling virtual function print with base-class"
43     << "\npointer to base-class object"
44     << "\ninvokes base-class print function:\n";
45 pointPtr->print();
46

```



```

47 // aim derived-class pointer at derived-class
48 // object and print
49 circlePtr = &circle;
50 cout << "\n\nCalling virtual function print with "
51     << "\nderived-class pointer to derived-class object "
52     << "\ninvokes derived-class print function:\n";
53 circlePtr->print();
54
55 // aim base-class pointer at derived-class object and print
56 pointPtr = &circle;
57 cout << "\n\nCalling virtual function print with base-class"
58     << "\npointer to derived-class object "
59     << "\ninvokes derived-class print function:\n";
60 pointPtr->print(); // polymorphism: invokes circle's print
61 cout << endl;
62
63 return 0;
64
65 } // 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.



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
```

Εικονικές Συναρτήσεις (Virtual Functions)

- Πολυμορφισμός
 - Το ίδιο μήνυμα, “print”, αποστέλλεται σε πολλά αντικείμενα
 - Τα αντικείμενα προσπελάζονται μέσω δείκτη της κλάσης βάσης
 - Το μήνυμα παίρνει “πολλές μορφές”
- Σύνοψη
 - Δείκτης κλάσης βάσης σε αντικείμενο της κλάσης βάσης, δείκτης παραγόμενης κλάσης σε αντικείμενο της παραγόμενης κλάσης
 - Απλό
 - Δείκτης κλάσης βάσης σε αντικείμενο της παραγόμενης κλάσης
 - Καλεί μόνο μεθόδους της κλάσης βάσης
 - Δεν ισχύει για τις μεταβλητές
 - Δείκτης παραγόμενης κλάσης σε αντικείμενο της κλάσης βάσης
 - Compiler error
 - Επιτρέπεται με downcasting

Παραδείγματα Πολυμορφισμού

- Παραδείγματα
 - Ας θεωρήσουμε ότι η κλάση **Rectangle** κληρονομεί την κλάση **Quadrilateral** (Τετράπλευρο)
 - Η **Rectangle** πιο ειδική της **Quadrilateral**
 - Οποιαδήποτε λειτουργία της **Quadrilateral** μπορεί να γίνει και για την κλάση **Rectangle** (i.e., perimeter, area)
- Εφαρμογή video game
 - Base class: **SpaceObject**
 - Derived: **Martian**, **SpaceShip**, **LaserBeam**
 - Κοινή συνάρτηση **draw**
 - Ζητούμενο: Η ανανέωση της οθόνης
 - Ο Screen Manager διαθέτει **vector** με base-class pointers σε αντικείμενα
 - Στέλνει **draw** μήνυμα σε κάθε αντικείμενο
 - Το ίδιο μήνυμα έχει «πολλές μορφές» αποτελεσμάτων

Παραδείγματα Πολυμορφισμού

- Εφαρμογή video game (συνέχεια)
 - Εύκολο να προσθέσουμε μια νέα κλάση **Mercurian**
 - Κληρονομεί την **SpaceObject**
 - Περιέχει τη δικιά της υλοποίηση για την **draw**
 - Ο Screen Manager δεν χρειάζεται αλλαγές στον κώδικά του
 - Καλεί την **draw** ανεξάρτητα από τον τύπο του αντικειμένου
 - **Mercurian** αντικείμενα ενσωματώνονται εύκολα
- Πολυμορφισμός
 - Χειρισμός αντικειμένων χωρίς να γνωρίζουμε τον τύπο τους
 - Επεκτάσιμα προγράμματα
 - Εύκολη προσθήκη κλάσεων

Χρήση πεδίου για τύπο αντικειμένων και εντολής `switch`

- Ένας τρόπος να προσδιορίσουμε την κλάση ενός αντικειμένου
 - Βάζουμε στην κλάση βάσης ένα πεδίο
 - `shapeType` στην κλάση `Shape`
 - Χρησιμοποιούμε μια `switch` για να καλέσουμε τη σωστή `print` συνάρτηση
- Πολλά προβλήματα
 - Μπορεί να ξεχάσουμε τον έλεγχο κάποιας περίπτωσης στην `switch`
 - Εάν προσθέσουμε/αφαιρέσουμε μια κλάση πρέπει να ενημερώσουμε την `switch`
 - Χρονοβόρο και επιρρεπές σε λάθη
- Καλύτερα με πολυμορφισμό
 - απλούστερα προγράμματα, λιγότερο debugging

Abstract Classes

- Abstract class
 - Σκοπός: να αποτελέσει μια base class (abstract base class)
 - Ελλιπής
 - Συμπληρώνεται με τις παραγόμενες κλάσεις
 - Δεν κατασκευάζουμε αντικείμενα από μια abstract class
 - Μπορούμε να έχουμε δείκτες και αναφορές
- Concrete classes
 - Μπορούμε να κατασκευάσουμε αντικείμενα
 - Υλοποιούμε όλες τις συναρτήσεις που περιέχουν

Abstract Classes

- Είναι χρήσιμες, όχι υποχρεωτικές
- Για να ορίσουμε μια abstract class
 - Θέλουμε μια ή περισσότερες "pure" virtual functions
 - `virtual void draw() const = 0;`
 - Regular virtual functions
 - Έχουν υλοποίηση, η υπερκάλυψη είναι προαιρετική
 - Pure virtual functions
 - Δεν έχουν υλοποίηση, η υπερκάλυψη είναι υποχρεωτική
 - Μια abstract class μπορεί να έχει δεδομένα και υλοποιημένες συναρτήσεις

Abstract Classes

- Δείκτες σε abstract base class
 - Χρήσιμοι για πολυμορφισμό
- Παράδειγμα εφαρμογής
 - Abstract class **Shape**
 - Ορίζει την **draw** ως pure virtual function
 - **Circle, Triangle, Rectangle** → **Shape**
 - Κάθε κλάση πρέπει να υλοποιήσει την **draw**
 - Ο Screen Manager γνωρίζει ότι κάθε αντικείμενο μπορεί να σχεδιάσει τον εαυτό του

Μελέτη Περίπτωσης : Κληρονóμηση Διεπαφής και Υλοποίησης

- Κάνουμε την **Shape** abstract base class
 - Pure virtual functions (πρέπει να συμπληρωθούν)
 - **getName, print**
 - Δεν έχει νόημα προκαθορισμένη υλοποίηση
 - Virtual functions (μπορούν να επανοριστούν)
 - **getArea, getVolume**
 - Αρχικά επιστρέφουν **0.0**
 - Εάν δεν επανοριστούν, χρησιμοποιούνται οι ορισμοί της base class
 - Παραγόμενες κλάσεις: **Point, Circle, Cylinder**

Μελέτη Περίπτωσης: Κληρονóμηση Διεπαφής και Υλοποίησης

	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



```

1 // Fig. 10.12: shape.h
2 // Shape abstract-base-class definition.
3 #ifndef SHAPE_H
4 #define SHAPE_H
5
6 #include <string> // C++ standard string class
7
8 using std::string;
9
10 class Shape {
11
12 public:
13
14     // virtual function that returns shape area
15     virtual double getArea() const;
16
17     // virtual function that returns shape volume
18     virtual double getVolume() const;
19
20     // pure virtual functions; overridden in derived classes
21     virtual string getName() const = 0; // return shape name
22     virtual void print() const = 0;    // output shape
23
24 }; // end class Shape
25
26 #endif

```

Virtual and pure virtual functions.



```

1 // Fig. 10.13: shape.cpp
2 // Shape class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "shape.h" // Shape class definition
8
9 // return area of shape; 0.0 by default
10 double getArea() const
11 {
12     return 0.0;
13 }
14 // end function getArea
15
16 // return volume of shape; 0.0 by default
17 double getVolume() const
18 {
19     return 0.0;
20 }
21 // end function getVolume

```



point.h (1 of 2)

```

1 // Fig. 10.14: point.h
2 // Point class definition represents an x-y coordinate pair.
3 #ifndef POINT_H
4 #define POINT_H
5
6 #include "shape.h" // Shape class definition
7
8 class Point : public Shape {
9
10 public:
11     Point( int = 0, int = 0 ); // default constructor
12
13     void setX( int ); // set x in coordinate pair
14     int getX() const; // return x from coordinate pair
15
16     void setY( int ); // set y in coordinate pair
17     int getY() const; // return y from coordinate pair
18
19     // return name of shape (i.e., "Point" )
20     virtual string getName() const;
21
22     virtual void print() const; // output Point object
23

```

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



point.h (2 of 2)

```

24 private:
25     int x; // x part of coordinate pair
26     int y; // y part of coordinate pair
27
28 }; // end class Point
29
30 #endif

```



```

1 // Fig. 10.15: point.cpp
2 // Point class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "point.h" // Point class definition
8
9 // default constructor
10 Point::Point( int xValue, int yValue )
11     : x( xValue ), y( yValue )
12 {
13     // empty body
14
15 } // end Point constructor
16
17 // set x in coordinate pair
18 void Point::setX( int xValue )
19 {
20     x = xValue; // no need for validation
21
22 } // end function setX
23

```



```

24 // return x from coordinate pair
25 int Point::getX() const
26 {
27     return x;
28
29 } // end function getX
30
31 // set y in coordinate pair
32 void Point::setY( int yValue )
33 {
34     y = yValue; // no need for validation
35
36 } // end function setY
37
38 // return y from coordinate pair
39 int Point::getY() const
40 {
41     return y;
42
43 } // end function getY
44

```



```

45 // override pure virtual function getName: return name of Point
46 string Point::getName() const
47 {
48     return "Point";
49 }
50 // end function getName
51
52 // override pure virtual function print: output Point object
53 void Point::print() const
54 {
55     cout << '[' << getX() << ", " << getY() << ']' ;
56 }
57 // end function print

```

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



```

1 // Fig. 10.16: circle.h
2 // Circle class contains x-y coordinate pair and radius.
3 #ifndef CIRCLE_H
4 #define CIRCLE_H
5
6 #include "point.h" // Point class definition
7
8 class Circle : public Point {
9
10 public:
11
12     // default constructor
13     Circle( int = 0, int = 0, double = 0.0 );
14
15     void setRadius( double ); // set radius
16     double getRadius() const; // return radius
17
18     double getDiameter() const; // return diameter
19     double getCircumference() const; // return circumference
20     virtual double getArea() const; // return area
21
22     // return name of shape (i.e., "Circle")
23     virtual string getName() const;
24
25     virtual void print() const; // output Circle object

```




```

26
27 private:
28     double radius; // Circle's radius
29
30 }; // end class Circle
31
32 #endif

```



```

1 // Fig. 10.17: circle.cpp
2 // Circle class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "circle.h" // Circle class definition
8
9 // default constructor
10 Circle::Circle( int xValue, int yValue, double radiusValue )
11     : Point( xValue, yValue ) // call base-class constructor
12 {
13     setRadius( radiusValue );
14
15 } // end Circle constructor
16
17 // set radius
18 void Circle::setRadius( double radiusValue )
19 {
20     radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
21
22 } // end function setRadius
23

```



circle.cpp (2 of 3)

```

24 // return radius
25 double Circle::getRadius() const
26 {
27     return radius;
28 }
29 // end function getRadius
30
31 // calculate and return diameter
32 double Circle::getDiameter() const
33 {
34     return 2 * getRadius();
35 }
36 // end function getDiameter
37
38 // calculate and return circumference
39 double Circle::getCircumference() const
40 {
41     return 3.14159 * getDiameter();
42 }
43 // end function getCircumference
44

```

© 2003 Prentice Hall, Inc.
All rights reserved.



circle.cpp (3 of 3)

```

45 // override virtual function getArea: return area of Circle
46 double Circle::getArea() const
47 {
48     return 3.14159 * getRadius() * getRadius();
49 }
50 // end function getArea
51
52 // override virtual function getName
53 string Circle::getName() const
54 {
55     return "Circle";
56 }
57 // end function getName
58
59 // override virtual function print: output Circle object
60 void Circle::print() const
61 {
62     cout << "center is ";
63     Point::print(); // invoke Point's print function
64     cout << "; radius is " << getRadius();
65 }
66 // end function print

```

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

© 2003 Prentice Hall, Inc.
All rights reserved.


cylinder.h (1 of 2)

```

1 // Fig. 10.18: cylinder.h
2 // Cylinder class inherits from class Circle.
3 #ifndef CYLINDER_H
4 #define CYLINDER_H
5
6 #include "circle.h" // Circle class definition
7
8 class Cylinder : public Circle {
9
10 public:
11
12     // default constructor
13     Cylinder( int = 0, int = 0, double = 0.0, double = 0.0 );
14
15     void setHeight( double ); // set Cylinder's height
16     double getHeight() const; // return Cylinder's height
17
18     virtual double getArea() const; // return Cylinder's area
19     virtual double getVolume() const; // return Cylinder's volume
20

```


cylinder.h (2 of 2)

```

21     // return name of shape (i.e., "Cylinder" )
22     virtual string getName() const;
23
24     virtual void print() const; // output Cylinder
25
26 private:
27     double height; // Cylinder's height
28
29 }; // end class Cylinder
30
31 #endif

```


cylinder.cpp
 (1 of 3)

```

1 // Fig. 10.19: cylinder.cpp
2 // Cylinder class inherits from class Circle.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "cylinder.h" // Cylinder class definition
8
9 // default constructor
10 Cylinder::Cylinder( int xValue, int yValue, double radiusValue,
11 double heightValue )
12 : Circle( xValue, yValue, radiusValue )
13 {
14     setHeight( heightValue );
15
16 } // end Cylinder constructor
17
18 // set Cylinder's height
19 void Cylinder::setHeight( double heightValue )
20 {
21     height = ( heightValue < 0.0 ? 0.0 : heightValue );
22
23 } // end function setHeight

```


cylinder.cpp
 (2 of 3)

```

24
25 // get Cylinder's height
26 double Cylinder::getHeight() const
27 {
28     return height;
29
30 } // end function getHeight
31
32 // override virtual function getArea: return Cylinder area
33 double Cylinder::getArea() const
34 {
35     return 2 * Circle::getArea() + // code reuse
36         getCircumference() * getHeight();
37
38 } // end function getArea
39
40 // override virtual function getVolume: return Cylinder volume
41 double Cylinder::getVolume() const
42 {
43     return Circle::getArea() * getHeight(); // code reuse
44
45 } // end function getVolume
46

```



cylinder.cpp
(3 of 3)

```

47 // override virtual function getName: return name of Cylinder
48 string Cylinder::getName() const
49 {
50     return "Cylinder";
51 }
52 } // end function getName
53
54 // output Cylinder object
55 void Cylinder::print() const
56 {
57     Circle::print(); // code reuse
58     cout << "; height is " << getHeight();
59 }
60 } // end function print

```

© 2003 Prentice Hall, Inc.
All rights reserved.



fig10_20.cpp
(1 of 5)

```

1 // Fig. 10.20: fig10_20.cpp
2 // Driver for shape, point, circle, cylinder hierarchy.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10
11 using std::setprecision;
12
13 #include <vector>
14
15 using std::vector;
16
17 #include "shape.h" // Shape class definition
18 #include "point.h" // Point class definition
19 #include "circle.h" // Circle class definition
20 #include "cylinder.h" // Cylinder class definition
21
22 void virtualViaPointer( const Shape * );
23 void virtualViaReference( const Shape & );
24

```

© 2003 Prentice Hall, Inc.
All rights reserved.



fig10_20.cpp
(2 of 5)

```

25 int main()
26 {
27     // set floating-point number format
28     cout << fixed << setprecision( 2 );
29
30     Point point( 7, 11 );           // create a Point
31     Circle circle( 22, 8, 3.5 );   // create a Circle
32     Cylinder cylinder( 10, 10, 3.3, 10 ); // create a Cylinder
33
34     cout << point.getName() << ": "; // static binding
35     point.print();                  // static binding
36     cout << '\n';
37
38     cout << circle.getName() << ": "; // static binding
39     circle.print();                // static binding
40     cout << '\n';
41
42     cout << cylinder.getName() << ": "; // static binding
43     cylinder.print();              // static binding
44     cout << "\n\n";
45

```

© 2003 Prentice Hall, Inc.
All rights reserved.



fig10_20.cpp
(3 of 5)

```

46     // create vector of three base-class pointers
47     vector< Shape * > shapeVector( 3 );
48
49     // aim shapeVector[0] at derived-class P
50     shapeVector[ 0 ] = &point;
51
52     // aim shapeVector[1] at derived-class C
53     shapeVector[ 1 ] = &circle;
54
55     // aim shapeVector[2] at derived-class C
56     shapeVector[ 2 ] = &cylinder;
57
58     // loop through shapeVector and call vir
59     // to print the shape name, attributes,
60     // of each object using dynamic binding
61     cout << "\nVirtual function calls made off "
62           << "base-class pointers:\n\n";
63
64     for ( int i = 0; i < shapeVector.size(); i++ )
65         virtualViaPointer( shapeVector[ i ] );
66

```

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.

© 2003 Prentice Hall, Inc.
All rights reserved.



```

67 // loop through shapeVector and call virtualViaReference
68 // to print the shape name, attributes, area and volume
69 // of each object using dynamic binding
70 cout << "\nVirtual function calls made
71     << "base-class references:\n\n";
72
73 for ( int j = 0; j < shapeVector.size(); j++ )
74     virtualViaReference( *shapeVector[ j ] );
75
76 return 0;
77
78 } // end main
79
80 // make virtual function calls off a base-class
81 // using dynamic binding
82 void virtualViaPointer( const Shape *baseClassPtr )
83 {
84     cout << baseClassPtr->getName() << ": ";
85
86     baseClassPtr->print();
87
88     cout << "\narea is " << baseClassPtr->getArea()
89         << "\nvolume is " << baseClassPtr->getVolume()
90         << "\n\n";
91
92 } // end function virtualViaPointer
93

```

Use references instead of pointers, for the same effect.

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



```

94 // make virtual function calls off a base-class reference
95 // using dynamic binding
96 void virtualViaReference( const Shape &baseClassRef )
97 {
98     cout << baseClassRef.getName() << ": ";
99
100    baseClassRef.print();
101
102    cout << "\narea is " << baseClassRef.getArea()
103        << "\nvolume is " << baseClassRef.getVolume() << "\n\n";
104
105 } // end function virtualViaReference

```

```
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
```



Outline



fig10_20.cpp
output (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

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
```



Outline

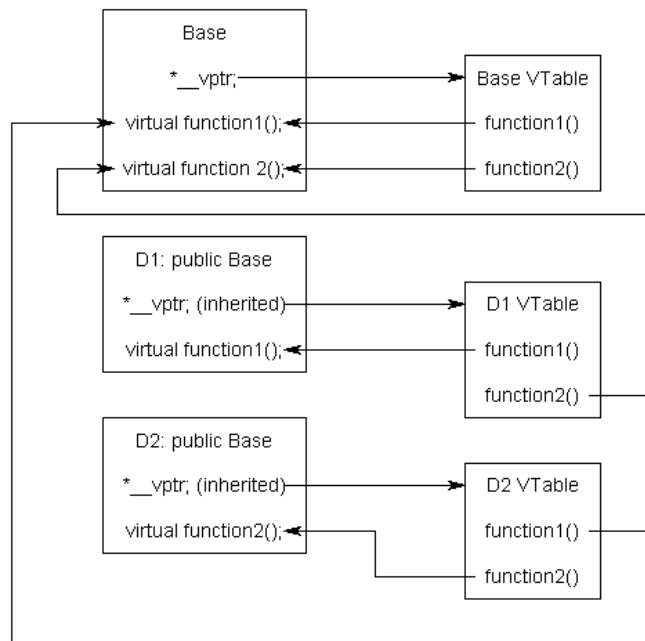


fig10_20.cpp
output (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Πολυμορφισμός, Εικονικές Συναρτήσεις και Δυναμική Σύνδεση: Υπάρχει κόστος χρήσης?

- Ο πολυμορφισμός έχει overhead
 - Δεν χρησιμοποιείται π.χ. στην STL (Standard Template Library) για λόγους επίδοσης
- **virtual** function table (vtable)
 - Κάθε κλάση με μια **virtual** function έχει ένα vtable
 - Για κάθε **virtual** function, το vtable έχει δείκτη στην κατάλληλη συνάρτηση



Virtual Destructors

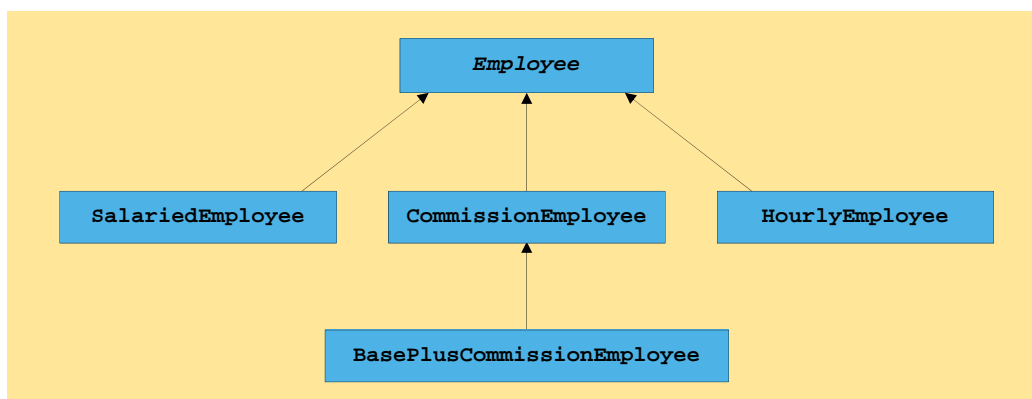
- Δείκτης κλάσης βάσης σε αντικείμενο της παραγόμενης κλάσης
 - Εάν καταστρέψουμε το αντικείμενο με την **delete**, η συμπεριφορά είναι απροσδιόριστη
- Απλή λύση
 - Δηλώνουμε τον destructor της base-class ως **virtual**
 - Όταν καλείται η **delete** καλείται και ο κατάλληλος destructor
- Όταν καταστρέφουμε ένα αντικείμενο μιας παραγόμενης κλάσης
 - Πρώτα εκτελείται ο destructor της παραγόμενης κλάσης
 - Μετά εκτελείται ο destructor της base-class
- Οι constructors δεν μπορεί να είναι **virtual**

Μελέτη Περίπτωσης: Σύστημα Μισθοδοσίας με Πολυμορφισμό

- Πρόγραμμα για μισθοδοσία
 - Χρήση virtual functions και πολυμορφισμού
- Πρόβλημα
 - 4 είδη εργαζομένων, αμείβονται σε εβδομαδιαία βάση
 - Salaried (μισθός, ανεξαρτήτως ωρών)
 - Hourly (υπερωρίες [>40 hours] + 50% το ωρομίσθιο)
 - Commission (ποσοστό επί των πωλήσεων)
 - Base-plus-commission (βασικός μισθός + ποσοστό επί των πωλήσεων)
 - Η διοίκηση επιθυμεί να δώσει αύξηση 10%

Μελέτη Περίπτωσης: Σύστημα Μισθοδοσίας με Πολυμορφισμό

- Base class: **Employee**
 - Pure virtual function **earnings** (returns pay)
 - Πρέπει να γνωρίζουμε το είδος του εργαζόμενου
 - Δεν έχει νόημα για τη γενική κλάση employee
 - Παραγόμενες κλάσεις από την **Employee**



Μελέτη Περίπτωσης: Σύστημα Μισθοδοσίας με Πολυμορφισμό

- Downcasting
 - `dynamic_cast` operator
 - Προσδιορίζει τον τύπο του αντικειμένου κατά τον χρόνο εκτέλεσης
 - Επιστρέφει 0 εάν δεν είναι σωστού τύπου (cannot be cast)

```
NewClass *ptr = dynamic_cast < NewClass *> objectPtr;
```
- Keyword `typeid`
 - Header `<typeinfo>`
 - Usage: `typeid(object)`
 - Επιστρέφει `type_info` object
 - Περιέχει πληροφορία για τον τύπο του ορίσματος, μαζί με το όνομά του
 - `typeid(object).name()`
 - Αντίστοιχο με `object.getClass().getName()` της Java

```

1 // Fig. 10.23: employee.h
2 // Employee abstract base class.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 #include <string> // C++ standard string class
7
8 using std::string;
9
10 class Employee {
11
12 public:
13     Employee( const string &, const string &, const string & );
14
15     void setFirstName( const string & );
16     string getFirstName() const;
17
18     void setLastName( const string & );
19     string getLastName() const;
20
21     void setSocialSecurityNumber( const string & );
22     string getSocialSecurityNumber() const;
23

```



Outline

70

employee.h (1 of 2)

**employee.h (2 of 2)**

```

24 // pure virtual function makes Employee abstract base class
25 virtual double earnings() const = 0; // pure virtual
26 virtual void print() const; // virtual
27
28 private:
29     string firstName;
30     string lastName;
31     string socialSecurityNumber;
32
33 }; // end class Employee
34
35 #endif // EMPLOYEE_H

```

**employee.cpp
(1 of 3)**

```

1 // Fig. 10.24: employee.cpp
2 // Abstract-base-class Employee member-function definitions.
3 // Note: No definitions are given for pure virtual functions.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include "employee.h" // Employee class definition
10
11 // constructor
12 Employee::Employee( const string &first, const string &last,
13     const string &SSN )
14     : firstName( first ),
15     lastName( last ),
16     socialSecurityNumber( SSN )
17 {
18     // empty body
19 }
20 // end Employee constructor
21

```



employee.cpp
(2 of 3)

```

22 // return first name
23 string Employee::getFirstName() const
24 {
25     return firstName;
26
27 } // end function getFirstName
28
29 // return last name
30 string Employee::getLastName() const
31 {
32     return lastName;
33
34 } // end function getLastName
35
36 // return social security number
37 string Employee::getSocialSecurityNumber() const
38 {
39     return socialSecurityNumber;
40
41 } // end function getSocialSecurityNumber
42
43 // set first name
44 void Employee::setFirstName( const string &first )
45 {
46     firstName = first;
47
48 } // end function setFirstName
49

```

© 2003 Prentice Hall, Inc.
All rights reserved.



employee.cpp
(3 of 3)

```

50 // set last name
51 void Employee::setLastName( const string &last )
52 {
53     lastName = last;
54
55 } // end function setLastName
56
57 // set social security number
58 void Employee::setSocialSecurityNumber( const string &number )
59 {
60     socialSecurityNumber = number; // should validate
61
62 } // end function setSocialSecurityNumber
63
64 // print Employee's information
65 void Employee::print() const
66 {
67     cout << getFirstName() << ' ' << getLastName()
68         << "\nsocial security number: "
69         << getSocialSecurityNumber() << endl;
70
71 } // end function print

```

Default implementation for
virtual function **print**.

© 2003 Prentice Hall, Inc.
All rights reserved.



salaried.h (1 of 1)

```

1 // Fig. 10.25: salaried.h
2 // SalariedEmployee class derived from Employee.
3 #ifndef SALARIED_H
4 #define SALARIED_H
5
6 #include "employee.h" // Employee class definition
7
8 class SalariedEmployee : public Employee {
9
10 public:
11     SalariedEmployee( const string &, const string &,
12                     const string &, double = 0.0 );
13
14     void setWeeklySalary( double );
15     double getWeeklySalary() const;
16
17     virtual double earnings() const;
18     virtual void print() const; // "salaried employee: "
19
20 private:
21     double weeklySalary;
22
23 }; // end class SalariedEmployee
24
25 #endif // SALARIED_H

```

New functions for the **SalariedEmployee** class. **earnings** must be overridden. **print** is overridden to specify that this is a salaried employee.



salaried.cpp (1 of 2)

```

1 // Fig. 10.26: salaried.cpp
2 // SalariedEmployee class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "salaried.h" // SalariedEmployee class definition
8
9 // SalariedEmployee constructor
10 SalariedEmployee::SalariedEmployee(
11     const string &last, const string &first,
12     double salary )
13     : Employee( first, last, socialSecurityNumber )
14 {
15     setWeeklySalary( salary );
16
17 } // end SalariedEmployee constructor
18
19 // set salaried employee's salary
20 void SalariedEmployee::setWeeklySalary( double salary )
21 {
22     weeklySalary = salary < 0.0 ? 0.0 : salary;
23
24 } // end function setWeeklySalary
25

```

Use base class constructor for basic fields.

salaried.cpp
(2 of 2)

```

26 // calculate salaried employee's pay
27 double SalariedEmployee::earnings() const
28 {
29     return getWeeklySalary();
30 }
31 } // end function earnings
32
33 // return salaried employee's salary
34 double SalariedEmployee::getWeeklySalary() const
35 {
36     return weeklySalary;
37 }
38 } // end function getWeeklySalary
39
40 // print salaried employee's name
41 void SalariedEmployee::print() const
42 {
43     cout << "\nsalaried employee: ";
44     Employee::print(); // code reuse
45 }
46 } // end function print

```

Must implement pure virtual functions.



hourly.h (1 of 1)

```

1 // Fig. 10.27: hourly.h
2 // HourlyEmployee class definition.
3 #ifndef HOURLY_H
4 #define HOURLY_H
5
6 #include "employee.h" // Employee class definition
7
8 class HourlyEmployee : public Employee {
9
10 public:
11     HourlyEmployee( const string &, const string &,
12                   const string &, double = 0.0, double = 0.0 );
13
14     void setWage( double );
15     double getWage() const;
16
17     void setHours( double );
18     double getHours() const;
19
20     virtual double earnings() const;
21     virtual void print() const;
22
23 private:
24     double wage; // wage per hour
25     double hours; // hours worked for week
26
27 }; // end class HourlyEmployee
28
29 #endif // HOURLY_H

```



hourly.cpp (1 of 3)

```

1 // Fig. 10.28: hourly.cpp
2 // HourlyEmployee class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "hourly.h"
8
9 // constructor for class HourlyEmployee
10 HourlyEmployee::HourlyEmployee( const string &first,
11     const string &last, const string &socialSecurityNumber,
12     double hourlyWage, double hoursWorked )
13     : Employee( first, last, socialSecurityNumber )
14 {
15     setWage( hourlyWage );
16     setHours( hoursWorked );
17
18 } // end HourlyEmployee constructor
19
20 // set hourly employee's wage
21 void HourlyEmployee::setWage( double wageAmount )
22 {
23     wage = wageAmount < 0.0 ? 0.0 : wageAmount;
24
25 } // end function setWage

```

© 2003 Prentice Hall, Inc.
All rights reserved.



hourly.cpp (2 of 3)

```

26
27 // set hourly employee's hours worked
28 void HourlyEmployee::setHours( double hoursWorked )
29 {
30     hours = ( hoursWorked >= 0.0 && hoursWorked <= 168.0 ) ?
31         hoursWorked : 0.0;
32
33 } // end function setHours
34
35 // return hours worked
36 double HourlyEmployee::getHours() const
37 {
38     return hours;
39
40 } // end function getHours
41
42 // return wage
43 double HourlyEmployee::getWage() const
44 {
45     return wage;
46
47 } // end function getWage
48

```

© 2003 Prentice Hall, Inc.
All rights reserved.



hourly.cpp (3 of 3)

```

49 // get hourly employee's pay
50 double HourlyEmployee::earnings() const
51 {
52     if ( hours <= 40 ) // no overtime
53         return wage * hours;
54     else // overtime is paid at wage * 1.5
55         return 40 * wage + ( hours - 40 ) * wage * 1.5;
56
57 } // end function earnings
58
59 // print hourly employee's information
60 void HourlyEmployee::print() const
61 {
62     cout << "\nhourly employee: ";
63     Employee::print(); // code reuse
64
65 } // end function print

```

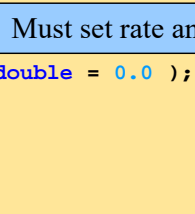
commission.h
(1 of 1)

```

1 // Fig. 10.29: commission.h
2 // CommissionEmployee class derived from Employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include "employee.h" // Employee class definition
7
8 class CommissionEmployee : public Employee {
9
10 public:
11     CommissionEmployee( const string &,
12                        const string &, double = 0.0, double = 0.0 );
13
14     void setCommissionRate( double );
15     double getCommissionRate() const;
16
17     void setGrossSales( double );
18     double getGrossSales() const;
19
20     virtual double earnings() const;
21     virtual void print() const;
22
23 private:
24     double grossSales; // gross weekly sales
25     double commissionRate; // commission percentage
26
27 }; // end class CommissionEmployee
28
29 #endif // COMMISSION_H

```

Must set rate and sales.




 commission.cpp
 (1 of 3)

```

1 // Fig. 10.30: commission.cpp
2 // CommissionEmployee class member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "commission.h" // Commission class
8
9 // CommissionEmployee constructor
10 CommissionEmployee::CommissionEmployee( const string &first,
11     const string &last, const string &socialSecurityNumber,
12     double grossWeeklySales, double percent )
13 : Employee( first, last, socialSecurityNumber )
14 {
15     setGrossSales( grossWeeklySales );
16     setCommissionRate( percent );
17
18 } // end CommissionEmployee constructor
19
20 // return commission employee's rate
21 double CommissionEmployee::getCommissionRate() const
22 {
23     return commissionRate;
24
25 } // end function getCommissionRate

```

© 2003 Prentice Hall, Inc.
 All rights reserved.


 commission.cpp
 (2 of 3)

```

26
27 // return commission employee's gross sales amount
28 double CommissionEmployee::getGrossSales() const
29 {
30     return grossSales;
31
32 } // end function getGrossSales
33
34 // set commission employee's weekly base salary
35 void CommissionEmployee::setGrossSales( double sales )
36 {
37     grossSales = sales < 0.0 ? 0.0 : sales;
38
39 } // end function setGrossSales
40
41 // set commission employee's commission
42 void CommissionEmployee::setCommissionRate( double rate )
43 {
44     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
45
46 } // end function setCommissionRate
47

```

© 2003 Prentice Hall, Inc.
 All rights reserved.



```

48 // calculate commission employee's earnings
49 double CommissionEmployee::earnings() const
50 {
51     return getCommissionRate() * getGrossSales();
52
53 } // end function earnings
54
55 // print commission employee's name
56 void CommissionEmployee::print() const
57 {
58     cout << "\ncommission employee: ";
59     Employee::print(); // code reuse
60
61 } // end function print
  
```



```

1 // Fig. 10.31: baseplus.h
2 // BasePlusCommissionEmployee class derived from Employee
3 #ifndef BASEPLUS_H
4 #define BASEPLUS_H
5
6 #include "commission.h" // Employee class definition
7
8 class BasePlusCommissionEmployee : public CommissionEmployee {
9
10 public:
11     BasePlusCommissionEmployee( const string &, const string &,
12         const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14     void setBaseSalary( double );
15     double getBaseSalary() const;
16
17     virtual double earnings() const;
18     virtual void print() const;
19
20 private:
21     double baseSalary; // base salary per week
22
23 }; // end class BasePlusCommissionEmployee
24
25 #endif // BASEPLUS_H
  
```

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


baseplus.cpp
 (1 of 2)

```

1 // Fig. 10.32: baseplus.cpp
2 // BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4
5 using std::cout;
6
7 #include "baseplus.h"
8
9 // constructor for class BasePlusCommissionEmployee
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last,
12     const string &socialSecurityNumber,
13     double grossSalesAmount, double rate,
14     double baseSalaryAmount )
15     : CommissionEmployee( first, last, socialSecurityNumber,
16       grossSalesAmount, rate )
17 {
18     setBaseSalary( baseSalaryAmount );
19 }
20 // end BasePlusCommissionEmployee constructor
21
22 // set base-salaried commission employee's wage
23 void BasePlusCommissionEmployee::setBaseSalary( double salary )
24 {
25     baseSalary = salary < 0.0 ? 0.0 : salary;
26 }
27 // end function setBaseSalary

```

© 2003 Prentice Hall, Inc.
 All rights reserved.


baseplus.cpp
 (2 of 2)

```

28
29 // return base-salaried commission employee's base salary
30 double BasePlusCommissionEmployee::getBaseSalary() const
31 {
32     return baseSalary;
33 }
34 // end function getBaseSalary
35
36 // return base-salaried commission employee's earnings
37 double BasePlusCommissionEmployee::earnings() const
38 {
39     return getBaseSalary() + CommissionEmployee::earnings();
40 }
41 // end function earnings
42
43 // print base-salaried commission employee's name
44 void BasePlusCommissionEmployee::print() const
45 {
46     cout << "\nbase-salaried commission employee: ";
47     Employee::print(); // code reuse
48 }
49 // end function print

```

© 2003 Prentice Hall, Inc.
 All rights reserved.

```

1 // Fig. 10.33: fig10_33.cpp
2 // Driver for Employee hierarchy.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10
11 using std::setprecision;
12
13 #include <vector>
14
15 using std::vector;
16
17 #include <typeinfo>
18
19 #include "employee.h" // Employee base class
20 #include "salaried.h" // SalariedEmployee class
21 #include "commission.h" // CommissionEmployee class
22 #include "baseplus.h" // BasePlusCommissionEmployee class
23 #include "hourly.h" // HourlyEmployee class
24

```



fig10_33.cpp
(1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

25 int main()
26 {
27     // set floating-point output formatting
28     cout << fixed << setprecision( 2 );
29
30     // create vector employees
31     vector < Employee * > employees( 4 );
32
33     // initialize vector with Employees
34     employees[ 0 ] = new SalariedEmployee( "John", "Smith",
35         "111-11-1111", 800.00 );
36     employees[ 1 ] = new CommissionEmployee( "Sue", "Jones",
37         "222-22-2222", 10000, .06 );
38     employees[ 2 ] = new BasePlusCommissionEmployee( "Bob",
39         "Lewis", "333-33-3333", 300, 5000, .04 );
40     employees[ 3 ] = new HourlyEmployee( "Karen", "Price",
41         "444-44-4444", 16.75, 40 );
42

```



fig10_33.cpp
(2 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

```

43 // generically process each element in
44 for ( int i = 0; i < employees.size();
45
46 // output employee information
47 employees[ i ]->print();
48
49 // downcast pointer
50 BasePlusCommissionEmployee *commissionPtr =
51     dynamic_cast < BasePlusCommissionEmployee * >
52     ( employees[ i ] );
53
54 // determine whether element points to base-salaried
55 // commission employee
56 if ( commissionPtr != 0 ) {
57     cout << "old base salary: $"
58         << commissionPtr->getBaseSalary() << endl;
59     commissionPtr->setBaseSalary(
60         1.10 * commissionPtr->getBaseSalary() );
61     cout << "new base salary with 10% increase is: $"
62         << commissionPtr->getBaseSalary() << endl;
63
64 } // end if
65
66 cout << "earned $" << employees[ i ]->earnings() << endl;
67
68 } // end for
69

```

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 **BasePlusCommissionEmployees**.

```

70 // release memory held by vector employees
71 for ( int j = 0; j < employees.size(); j++ ) {
72
73 // output class name
74 cout << "\ndeleting object of "
75     << typeid( *employees[ j ] ).name();
76
77 delete employees[ j ];
78
79 } // end for
80
81 cout << endl;
82
83 return 0;
84
85 } // end main

```

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



Outline

fig10_33.cpp
(4 of 4)

**fig10_33.cpp**
output (1 of 1)

```
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
```

© 2003 Prentice Hall, Inc.
All rights reserved.

Αναφορές

- Harvey M. Deitel, Paul J. Deitel, C++ How to Program, 4th Edition, Prentice Hall.
- Bjarne Stroustrup, The C++ Programming Language, Special Edition, Addison-Wesley.