IS 0020
Program Design and Software Tools

Polymorphism
Lecture 8

September 28/29, 2004

Introduction

- Polymorphism
  - "Program in the general"
  - Derived-class object can be treated as base-class object
    - "is-a" relationship
    - Base class is not a derived class object
  - Virtual functions and dynamic binding
  - Makes programs extensible
    - New classes added easily, can still be processed
- Examples
  - Use abstract base class **Shape**
    - Defines common interface (functionality)
    - **`Point`**, **`Circle`** and **`Cylinder`** inherit from **`Shape`**

Invoking Base-Class Functions from Derived-Class Objects

- Pointers to base/derived objects
  - Base pointer aimed at derived object
    - "is a" relationship
      - **`Circle`** "is a" **`Point`**
    - Will invoke base class functions
  - Can cast base-object's address to derived-class pointer
    - Called down-casting
    - Allows derived-class functionality
- Key point
  - Base-pointer can aim at derived-object - but can only call base-class functions
    - Data type of pointer/reference determines functions it can call


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

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

#include <iomanip>

using std::setprecision;

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

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

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

```

5

Outline

fig10_05.cpp
(2 of 3)

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

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

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

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

```

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

6

Outline

fig10_05.cpp
(3 of 3)

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

   return 0;

} // end main
```

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

7

Outline

fig10_06.cpp
(1 of 1)

fig10_06.cpp
output (1 of 1)

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

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

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

   return 0;

} // end main
```

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


8

Outline

fig10_07.cpp
(1 of 2)

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

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

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

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

```

Outline

fig10_07.cpp
(2 of 2)

```
   // attempt to invoke derived-class-only member functions
   // on derived-class object through base-class pointer
   double radius = pointPtr->getRadius();
   pointPtr->setRadius( 33.33 );
   double diameter = pointPtr->getDiameter();
   double circumference = pointPtr->getCircumference();
   double area = pointPtr->getArea();

   return 0;

} // end main
```

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

Virtual Functions

- **`virtual`** functions
  - Object (not pointer) determines function called
- Why useful?
  - Suppose **`Circle`**, **`Triangle`**, **`Rectangle`** derived from **`Shape`**
    - Each has own **`draw`** function
  - To draw any shape
    - Have base class **`Shape`** pointer, call **`draw`**
    - Program determines proper **`draw`** function at run time (dynamically)
    - Treat all shapes generically

Virtual Functions

- Declare **`draw`** as **`virtual`** in base class
  - Override **`draw`** in each derived class
    - Like redefining, but new function must have same signature
  - If function declared **`virtual`**, can only be overridden
    - **`virtual void draw() const;`**
    - Once declared **`virtual`**, **`virtual`** in all derived classes
      - Good practice to explicitly declare **`virtual`**
- Dynamic binding
  - Choose proper function to call at run time
  - Only occurs off pointer handles
    - If function called from object, uses that object's definition

Virtual Functions

- Polymorphism
  - Same message, "print", given to many objects
    - All through a base pointer
  - Message takes on "many forms"
- Summary
  - Base-pointer to base-object, derived-pointer to derived
    - Straightforward
  - Base-pointer to derived object
    - Can only call base-class functions
  - Derived-pointer to base-object
    - Compiler error
    - Allowed if explicit cast made

13

Polymorphism Examples

- Suppose designing video game
  - Base class **`SpaceObject`**
    - Derived **`Martian`**, **`SpaceShip`**, **`LaserBeam`**
    - Base function **`draw`**
  - To refresh screen
    - Screen manager has **`vector`** of base-class pointers to objects
    - Send **`draw`** message to each object
    - Same message has "many forms" of results
  - Easy to add class **`Mercurian`**
    - Inherits from **`SpaceObject`**
    - Provides own definition for **`draw`**
  - Screen manager does not need to change code
    - Calls **`draw`** regardless of object's type
    - **`Mercurian`** objects "plug right in"

14

Type Fields and switch Structures

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

15

Abstract Classes

- Abstract classes
  - Sole purpose: to be a base class (called abstract base classes)
  - Incomplete
    - Derived classes fill in "missing pieces"
  - Cannot make objects from abstract class
    - However, can have pointers and references
- Concrete classes
  - Can instantiate objects
  - Implement all functions they define
  - Provide specifics

16

Abstract Classes

- Abstract classes not required, but helpful
- To make a class abstract
  - Need one or more "pure" virtual functions
    - Declare function with initializer of 0

```
virtual void draw() const = 0;
```

  - Regular virtual functions
    - Have implementations, overriding is optional
  - Pure virtual functions
    - No implementation, must be overridden
  - Abstract classes can have data and concrete functions
    - Required to have one or more pure virtual functions

Case Study: Inheriting Interface and Implementation

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

Case Study: Inheriting Interface and Implementation

	getArea	getVolume	getName	print
Shape	0.0	0.0	= 0	= 0
Point	0.0	0.0	"Point"	[x,y]
Circle	πr^2	0.0	"Circle"	center=[x,y]; radius=r
Cylinder	$2\pi r^2 + 2\pi rh$	$\pi r^2 h$	"Cylinder"	center=[x,y]; radius=r; height=h


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

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

using std::string;

class Shape {

public:

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

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

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

}; // end class Shape

#endif
```

Virtual and pure virtual functions.


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

using std::cout;

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

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

}  // end function getArea

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

}  // end function getVolume
```

21

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- Polymorphism has overhead
  - Not used in STL (Standard Template Library) to optimize performance
- **`virtual`** function table (vtable)
  - Every class with a **`virtual`** function has a vtable
  - For every **`virtual`** function, vtable has pointer to the proper function
  - If derived class has same function as base class
    - Function pointer aims at base-class function

22

Virtual Destructors

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

23

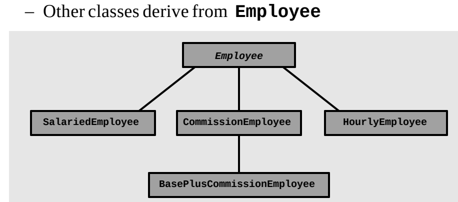
Case Study: Payroll System Using Polymorphism

- Create a payroll program
  - Use virtual functions and polymorphism
- Problem statement
  - 4 types of employees, paid weekly
    - Salaried (fixed salary, no matter the hours)
    - Hourly (overtime [>40 hours] pays time and a half)
    - Commission (paid percentage of sales)
    - Base-plus-commission (base salary + percentage of sales)
      - Boss wants to raise pay by 10%

24

Payroll System Using Polymorphism

- Base class **`Employee`**
  - Pure virtual function **`earnings`** (returns pay)
    - Pure virtual because need to know employee type
    - Cannot calculate for generic employee
  - Other classes derive from **`Employee`**



Dynamic Cast

- Downcasting
  - **`dynamic_cast`** operator
    - Determine object's type at runtime
    - Returns 0 if not of proper type (cannot be cast)

```
NewClass *ptr = dynamic_cast < NewClass *> objectPtr;
```

- Keyword **`typeid`**
  - Header **`<typeinfo>`**
  - Usage: **`typeid(object)`**
    - Returns **`type_info`** object
    - Has information about type of operand, including name
    - **`typeid(object).name()`**

Outline

employee.h (1 of 2)

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

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

using std::string;

class Employee {

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

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

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

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

```


Outline

employee.h (2 of 2)

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

private:
   string firstName;
   string lastName;
   string socialSecurityNumber;

}; // end class Employee

#endif // EMPLOYEE_H
```


Outline

employee.cpp
(1 of 3)

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

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

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

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

} // end Employee constructor

```

Outline

employee.cpp
(2 of 3)

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

} // end function getFirstName

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

} // end function getLastName

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

} // end function getSocialSecurityNumber

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

} // end function setFirstName

```


Outline

employee.cpp
(3 of 3)

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

} // end function setLastName

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

} // end function setSocialSecurity

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

} // end function print
```

Default implementation for virtual function **print**.

Outline

salaried.h (1 of 1)

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

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

class SalariedEmployee : public Employee {

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

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

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

private:
   double weeklySalary;

}; // end class SalariedEmployee

#endif // SALARIED_H
```

New functions for the **SalariedEmployee** class.

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

Outline

salaried.cpp
(1 of 2)

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

using std::cout;

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

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

} // end SalariedEmployee constructor

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

} // end function setWeeklySalary

```

Use base class constructor for basic fields.

Outline

salaried.cpp
(2 of 2)

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

} // end function earnings

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

} // end function getWeeklySalary

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

} // end function print
```

Must implement pure virtual functions.

Outline

hourly.h (1 of 1)

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

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

class HourlyEmployee : public Employee {

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

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

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

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

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

}; // end class HourlyEmployee

#endif // HOURLY_H
```


Outline

hourly.cpp (1 of 3)

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

using std::cout;

#include "hourly.h"

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

} // end HourlyEmployee constructor

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

} // end function setWage
```


Outline

hourly.cpp (2 of 3)

```

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

} // end function setHours

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

} // end function getHours

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

} // end function getWage

```

37

Outline

hourly.cpp (3 of 3)

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

} // end function earnings

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

} // end function print
```


38

Outline

commission.h (1 of 1)

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

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

class CommissionEmployee : public Employee {

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

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

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

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

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

}; // end class CommissionEmployee

#endif  // COMMISSION_H
```

Must set rate and sales.

39

Outline

commission.cpp (1 of 3)

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

using std::cout;

#include "commission.h"  // Commission class

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

} // end CommissionEmployee constructor

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

} // end function getCommissionRate
```


40

Outline

commission.cpp (2 of 3)

```

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

} // end function getGrossSales

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

} // end function setGrossSales

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

} // end function setCommissionRate

```

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

} // end function earnings

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

} // end function print
```



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

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

class BasePlusCommissionEmployee : public CommissionEmployee {

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

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

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

private:
   double baseSalary;      // base salary per week

}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

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


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

using std::cout;

#include "baseplus.h"

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

} // end BasePlusCommissionEmployee constructor

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

} // end function setBaseSalary
```



```

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

} // end function getBaseSalary

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

} // end function earnings

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

} // end function print
```

Outline

fig10_33.cpp
(1 of 4)

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

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

#include <iomanip>

using std::setprecision;

#include <vector>

using std::vector;

#include <typeinfo>

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

```


Outline

fig10_33.cpp
(2 of 4)

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

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

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

```



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

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

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

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

      } // end if

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

   } // end for

```

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

Outline

fig10_33.cpp
(4 of 4)

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

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

      delete employees[ j ];

   } // end for

   cout << endl;

   return 0;

} // end main
```

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

Outline

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
```