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 \texttt{Shape}
    • Defines common interface (functionality)
    • \texttt{Point, Circle, and Cylinder} inherit from \texttt{Shape}

Invoking Base-Class Functions from Derived-Class Objects

• Pointers to base/derived objects
  − Base pointer aimed at derived object
    • “is-a” relationship
      − \texttt{Circle} “is-a” \texttt{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

Outline

\texttt{fig10_05.cpp}

\begin{verbatim}
#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

  // Fig. 10.6: fig10_06.cpp
  // Handling base-class and derived-class pointers at base-class level
  // and derived-class objects, respectively.

  point.printPoint();// Point class function
  circle.printCircle(); // Circle class function
}
\end{verbatim}
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.

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

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

// aim base-class pointer at base-class object and print
pointPtr = &point;
cout << "

Calling print with base-class pointer to 
base-class object invokes base-class print 
function:
";
pointPtr->print();  // invokes Point's print

// aim derived-class pointer at derived-class object
// and print
circlePtr = &circle;
cout << "

Calling print with derived-class pointer to 
derived-class object invokes derived-class 
print function:
";
circlePtr->print();  // invokes Circle's print

// aim base-class pointer at derived-class object and print
pointPtr = &circle;
cout << "

Calling print with base-class pointer to 
derived-class object
invokes base-class print function on that derived-class object:
";
pointPtr->print(); // invokes Point's print
```

Aiming a base-class pointer at a derived class object is not 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.

```cpp
// 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;
}
```

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.

```cpp
// Fig. 10.7: fig10_07.cpp
// Attempting to invoke derived-class-only member functions
#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
    int x = pointPtr->getX();
    int y = pointPtr->getY();
    pointPtr->setX( 10 );
    pointPtr->setY( 10 );
    pointPtr->print();

    return 0;
}
```

// invoke base-class member functions on derived-class
// through a base-class pointer.
```
```
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

• 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`, 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

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
Polymorphism Examples

Suppose designing video game
- Base class `SpaceObject`
  - Derived `Martian`, `SpaceShip`, `LaserBeam`
  - Base function `draw`
- To refresh screen
  - Screen manager has a 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"

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

Abstract Classes

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

Abstract Classes

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

```
// 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 double getArea() const; // virtual function that returns shape area
    virtual double getVolume() const; // virtual function that returns shape volume
    virtual string getName() const = 0; // return shape name
    virtual void print() const = 0;     // output shape

}; // end class Shape

#endif
```

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

double getArea() const
{
    return 0.0;
}

double getVolume() const
{
    return 0.0;
}
```
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

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

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%

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

Diagram:

- Employee
  - SalariedEmployee
  - CommissionEmployee
  - HourlyEmployee
  - BasePlusCommissionEmployee
Dynamic Cast

- **Downcasting**
  - `dynamic_cast` operator
  - Determine object's type at runtime
  - Returns 0 if not of proper type (cannot be cast)
  
  ```cpp
  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
  
  ```cpp
  typeid(object).name();
  ```
```cpp
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
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;
61 }
62 // end function setSocialSecurityNumber
63
64 // print Employee's information
65 void Employee::print() const
66 {
67     cout << getFirstName() << ' ' << getLastName() << '
68         << "social security number: " << getSocialSecurityNumber() << endl;
69 }
70 // end function print
71

Outline
employee.cpp
(3 of 3)

Outline
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 public:
10     SalariedEmployee(const string &, const string &, const string &, double = 0.0);
11     void setWeeklySalary(double);
12     double getWeeklySalary() const;
13     virtual double earnings() const;
14     virtual void print() const;
15 private:
16     double weeklySalary;
17 }; // end class SalariedEmployee
18
19 #endif // SALARIED_H
20

Outline
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(const string &first, const string &last, const string &socialSecurityNumber, double salary)
11     : Employee(first, last, socialSecurityNumber) {
12     setWeeklySalary(salary);
13 } // end SalariedEmployee constructor
14
15 // set weekly salary
16 void SalariedEmployee::setWeeklySalary(double salary)
17 {
18     weeklySalary = salary < 0.0 ? 0.0 : salary;
19 } // end function setWeeklySalary
20
21 SalariedEmployee::~SalariedEmployee() {
22 } // end function ~SalariedEmployee
23
24 // virtual function earnings
25 double SalariedEmployee::earnings() const
26 {
27     return weeklySalary;
28 } // end function earnings
29
30 // virtual function print
31 void SalariedEmployee::print() const
32 {
33     cout << "salaried employee: ";
34     Employee::print();
35 } // end function print
36
37 // Use base class constructor for basic fields.
38 // New functions for the SalariedEmployee class
39 // warnings must be overridden to specify that this
40 // is a salaried employee.
26 // calculate salaried employee's pay
27 double SalariedEmployee::earnings() const
28 {
29     return getWeeklySalary();
30 } // end function earnings
31
32 // return salaried employee's salary
33 double SalariedEmployee::getWeeklySalary() const
34 {
35     return weeklySalary;
36 } // end function getWeeklySalary
37
38 // print salaried employee's name
39 void SalariedEmployee::print() const
40 {
41     cout << "salaried employee: ";
42     Employee::print(); // code reuse
43 } // end function print

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     void setWage( double );
14     double getWage() const;
15     void setHours( double );
16     double getHours() const;
17     virtual double earnings() const;
18     virtual void print() const;
19
20 private:
21     double wage;   // wage per hour
22     double hours; // hours worked for week
23
24     HourlyEmployee( const string &, const string &, const string &, double, double );
25
26 #endif // HOURLY_H

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 } // end HourlyEmployee constructor
18
19 // set hourly employee's wage
20 void HourlyEmployee::setWage( double wageAmount )
21 {
22     wage = wageAmount < 0.0 ? 0.0 : wageAmount;
23 } // end function setWage
24
25 // set hourly employee's hours worked
26 void HourlyEmployee::setHours( double wageAmount )
27 {
28     hours = ( hoursWorked >= 0.0 && hoursWorked <= 168.0 ) ?
29             hoursWorked : 0.0;
30 } // end function setHours
31
32 // return hours worked
33 double HourlyEmployee::getHours() const
34 {
35     return hours;
36 } // end function getHours
37
38 // return wage
39 double HourlyEmployee::getWage() const
40 {
41     return wage;
42 } // end function getWage
// 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;
}

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

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

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

class CommissionEmployee : public Employee {

public:
    CommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0 );
    void setCommissionRate( double );
    double getCommissionRate() const;
    void setGrossSales( double );
    double getGrossSales() const;
    virtual double earnings() const;
    virtual void print() const;

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

}; // end class CommissionEmployee
#endif // COMMISSION_H

// Fig. 10.30: commission.cpp
// CommissionEmployee class member-function definitions.

#include <iostream>
#include "commission.h" // Commission class

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

double CommissionEmployee::getCommissionRate() const
{ return commissionRate;
}

double CommissionEmployee::getGrossSales() const
{ return grossSales;
}

void CommissionEmployee::setGrossSales( double sales )
{ grossSales = sales < 0.0 ? 0.0 : sales;
}

void CommissionEmployee::setCommissionRate( double rate )
{ commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
}
```
48 // calculate commission employee's earnings
49 double CommissionEmployee::earnings() const
50 {
51     return getCommissionRate() * getGrossSales();
52 }
```

```
1 // Fig. 10.31: baseplus.h
2 // BasePlusCommissionEmployee class derived from CommissionEmployee.
3 #ifndef BASEPLUS_H
4 #define BASEPLUS_H
5
6 #include "commission.h" // Employee class definition
7
8 class BasePlusCommissionEmployee : public CommissionEmployee {
9     public:
10         BasePlusCommissionEmployee( const string &, const string &,
11             const string &, double = 0.0, double = 0.0, double = 0.0 );
12         void setBaseSalary( double );
13         double getBaseSalary() const;
14         virtual double earnings() const;
15         virtual void print() const;
16     private:
17         double baseSalary;      // base salary per week
18     };
19
20 #endif // BASEPLUS_H
```
Use downcasting to cast the employee object into a \texttt{BasePlusCommissionEmployee}. If it points to the correct type of object, the pointer is non-zero. This way, we can give a raise to only \texttt{BasePlusCommissionEmployee}s.

The \texttt{typeid} function returns a \texttt{type_info} object. This object contains information about the operand, including its name.
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