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

```
#include <iostream>
#include <iomanip>

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

#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
```
// set floating-point numeric format
cout << fixed << setprecision ( 2 );
// output objects point and circle
cout << "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: \n";
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: \n";
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: \n";
pointPtr->print();  // invokes Point's print

cout << endl;
return 0;
}  // end main

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. 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.
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 int main()
6 {
7   Point point( 30, 50 );
8   Circle *circlePtr = 0;
9   // aim derived-class pointer at base-class object
10   circlePtr = &point; // Error: a Point is not a Circle
11   return 0;
12 }

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

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 int main()
7 {
8   Point *pointPtr = 0;
9   Circle circle( 120, 89, 2.7 );
10   // aim base-class pointer at derived-class object
11   pointPtr = &circle;
12   // invoke base-class member functions on derived-class
13   // object through base-class pointer
14   int x = pointPtr->getX();
15   int y = pointPtr->getY();
16   pointPtr->setX( 10 );
17   pointPtr->setY( 10 );
18   pointPtr->print();
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
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”

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
      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**

### Table: Inheriting Interface and Implementation

<table>
<thead>
<tr>
<th></th>
<th><code>getArea</code></th>
<th><code>getVolume</code></th>
<th><code>getName</code></th>
<th><code>print</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape</td>
<td>0.0</td>
<td>0.0</td>
<td>= 0</td>
<td>= 0</td>
</tr>
<tr>
<td>Point</td>
<td>0.0</td>
<td>0.0</td>
<td>&quot;Point&quot;</td>
<td>[x, y]</td>
</tr>
<tr>
<td>Circle</td>
<td>πr²</td>
<td>0.0</td>
<td>&quot;Circle&quot;</td>
<td>center=[x, y]; radius=r</td>
</tr>
<tr>
<td>Cylinder</td>
<td>2πr² + 2πrh</td>
<td>πr²h</td>
<td>&quot;Cylinder&quot;</td>
<td>center=[x, y]; radius=r; height=h</td>
</tr>
</tbody>
</table>
// 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  
  string getName() const = 0;  // return shape name  
  void print() const = 0;     // output shape
}; // end class Shape
#endif

// Fig. 10.13: shape.cpp  
// Shape class member-function definitions.  
#include <iostream>
using std::cout;
#include "shape.h" // Shape class definition
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
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
Dynamic Cast

• Downcasting
  - `dynamic_cast` operator
    • Determine object's type at runtime
    • Returns 0 if not of proper type (cannot be cast)
    
    \[
    \text{NewClass }^* \text{ptr} = \text{dynamic_cast }< \text{NewClass }^* > \text{ objectPtr;}
    \]

• Keyword `typeid`
  – Header `<typeinfo>`
  – Usage: `typeid(object)`
    • Returns `type_info` object
    • Has information about type of operand, including name
    
    \[
    \text{typeid(object).name()}
    \]
// 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

#include <iostream>
#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
}

#endif // EMPLOYEE_H

#include <iostream>
#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
}

#endif // EMPLOYEE_H
```cpp
// 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

// 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 setSocialSecurityNumber

// 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`. 

class SalariedEmployee : public Employee {

public:

SalariedEmployee( const string &, const string &, 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

// SalariedEmployee constructor
SalariedEmployee::SalariedEmployee ( const string &first,
const string &last, const string &socialSecurityNumber,
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

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

Use base class constructor for basic fields.
```cpp
// calculate salaried employee's pay
26 double SalariedEmployee::earnings() const
27 {
28 return getWeeklySalary();
29 } // end function earnings
30 // return salaried employee's salary
33 double SalariedEmployee::getWeeklySalary() const
34 { return weeklySalary; }
35 } // end function getWeeklySalary
36 // print salaried employee's name
40 void SalariedEmployee::print() const
41 { cout << "salaried employee: ";
42 Employee::print(); // code reuse
43 } // end function print
```

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

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

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

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

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

```cpp
// 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>  
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  

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

1 // Fig. 10.31: baseplus.h
2 // BasePlusCommissionEmployee class derived from Employee.
3 #ifndef BASEPLUS_H
4 #define BASEPLUS_H
5 #include "commission.h" // Employee class definition
6
7 class BasePlusCommissionEmployee : public CommissionEmployee {
8
9 public:
10     BasePlusCommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0, double = 0.0 );
11     void setBaseSalary( double );
12     double getBaseSalary() const;
13     virtual double earnings() const;
14     virtual void print() const;
15
16 private:
17     double baseSalary; // base salary per week
18 }; // end class BasePlusCommissionEmployee
19
20 #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 << "\base-salaried commission employee: ";
    Employee::print(); // code reuse
} // end function print
```cpp
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 );
```
// generically process each element
for ( int i = 0; i < employees.size(); i++ ) {
    // 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

// 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

// return 0;
return 0;
} // end main

typeid returns a 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