Cpt S 122 – Data Structures

Polymorphism

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Introduction
- Introduction to Polymorphism
- Relationship among Objects in Inheritance Hierarchy
- Abstract Classes & pure `virtual` Functions
- Polymorphic processing
- `virtual` Functions & Dynamic Binding
- Polymorphism & Runtime Type Information (RTTI)
  - downcasting, dynamic_cast, typeid, type_info
- `virtual` Destructors

Introduction

- One name, multiple forms
- Have we seen polymorphism before?
  - Overloaded function, overloaded operators
  - Overloaded member functions are selected for invoking by matching argument, both *type and number*
  - Information is known to the compiler at *compile time*
    - Compiler is able to select the appropriate function at the compile time
  - This is called *early binding, or static binding, or static linking*
    - An object is bound to its function call at compile time
  - This is also known as *compile time polymorphism*

Introduction (cont.)

- Consider the following class definition where the function name and prototype is same in both the base and derived classes.

```
class A{
        int x;
        public:
          void show() {…} //show() in base class
};
class B: public A{
        int y;
        public:
          void show() {…} //show() in derived class
};
```

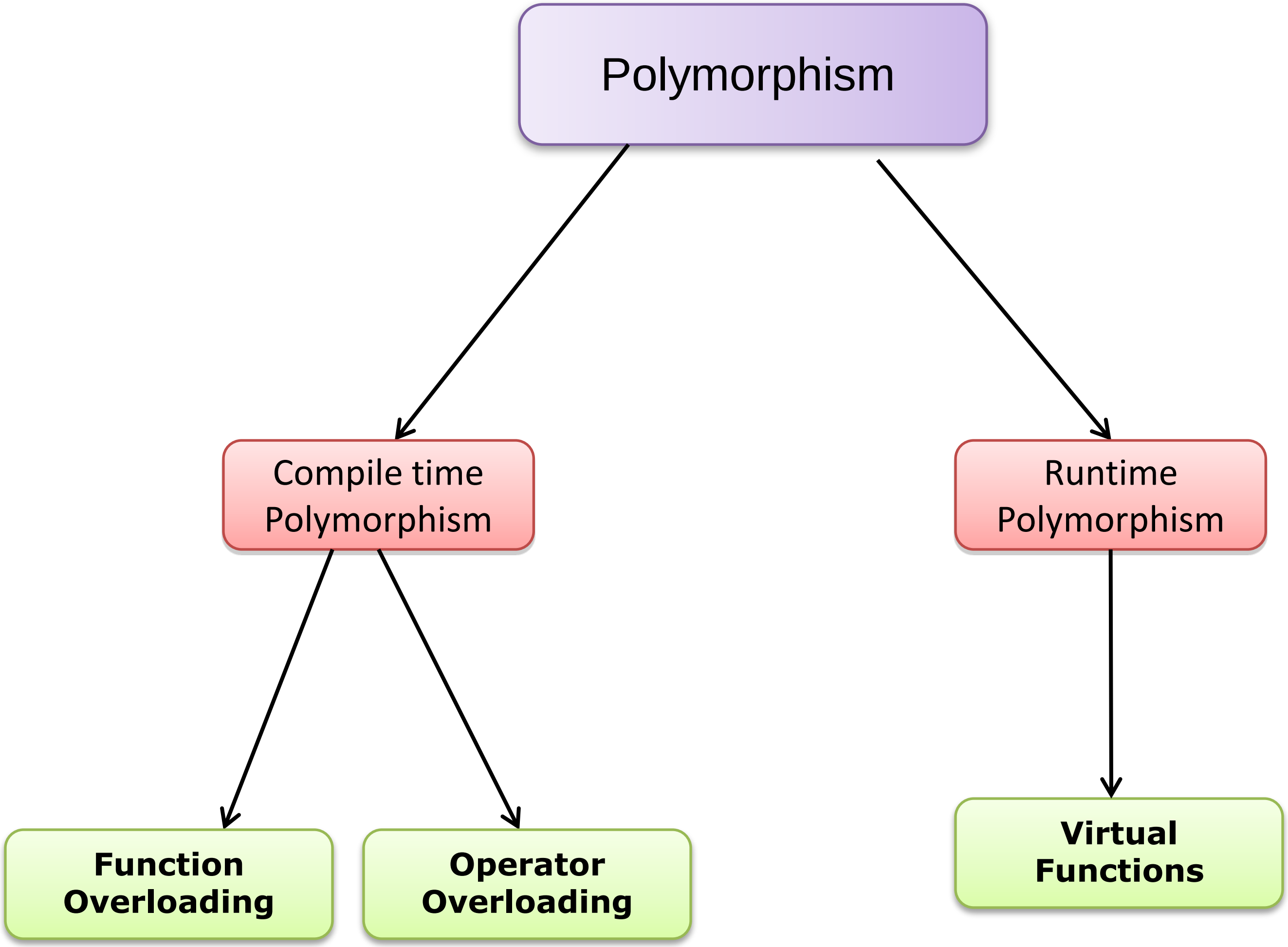
Introduction (cont.)

- How do we use the member function show() to print the values of objects of both the classes A and B?
  - prototype show() is same in both the places.
  - The function is not overloaded and therefore static binding does not apply.
- It would be nice if appropriate member function could be selected while the program is running
  - This is known as runtime polymorphism
  - How could it happen?
    - C++ supports a mechanism known as virtual function to achieve runtime polymorphism
    - At run time, when it is known what class objects are under consideration, the appropriate version of the function is called.

Introduction (cont.)

- Function is linked with a particular class much later after the compilation, this processed is termed as *late binding*
  - It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at runtime.
- Dynamic binding is one of the powerful feature in C++
  - Requires the use of pointers to objects
  - We will discuss in detail how the object pointers and virtual functions are used to implement dynamic binding or runtime polymorphism

Introduction (cont.)



Introduction (cont.)

- Polymorphism enables us to “program in the general” rather than “program in the specific.”
  - Enables us to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy’s base class.
- Polymorphism works off
  - base-class pointer handles
  - base-class reference handles
  - but not off name handles.
- Relying on each object to know how to “do the right thing” in response to the same function call is the key concept of polymorphism.
  - The same message sent to a variety of objects has “many forms” of results
- Polymorphism is the ability to create a variable, a function, or an object that has more than one form.

Introduction (cont.)

- With polymorphism, we can design and implement systems that are easily extensible.
  - New classes can be added with little or no modification to the general portions of the program
  - New types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system.
  - Client code that instantiate new objects must be modified to accommodate new types.
- Direct a variety of objects to behave in manners appropriate to those objects without even knowing their types
  - Those objects belong to the same inheritance hierarchy and are being accessed off a common base class pointer or common base class reference.

Relationships Among Objects in an Inheritance Hierarchy

- Demonstrate how base-class and derived-class pointers can be aimed at base-class and derived-class objects
  - how those pointers can be used to invoke member functions that manipulate those objects.
- A key concept
  - an object of a derived class can be treated as an object of its base class.
  - the compiler allows this because each derived-class object *is an* object of its base class.
- However, we cannot treat a base-class object as an object of any of its derived classes.
- The *is-a* relationship applies only from a derived class to its direct and indirect base classes.

Invoking Base-Class Functions from Derived-Class Objects

- Example classes: `CommissionEmployee` and `BasePlusCommissionEmployee`
- Aim a base-class pointer at a base-class object
  - invoke base-class functionality.
- Aim a derived-class pointer at a derived-class object
  - invoke derived-class functionality.
- Relationship between derived classes and base classes (i.e., the *is-a* relationship of inheritance)
  - aiming a base-class pointer at a derived-class object.
  - the base-class functionality is indeed available in the derived-class object.

```cpp
// Fig. 13.1: fig13_01.cpp
// Aiming base-class and derived-class pointers at base-class
// and derived-class objects, respectively.
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

```

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 5.)

```
   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // output objects commissionEmployee and basePlusCommissionEmployee
   cout << "Print base-class and derived-class objects:\n\n";
   commissionEmployee.print(); // invokes base-class print
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // invokes derived-class print

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee; // perfectly natural
   cout << "\n\n\nCalling print with base-class pointer to "
      << "\nbase-class object invokes base-class print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

```

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 5.)

```
   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
   cout << "\n\n\nCalling print with derived-class pointer to "
      << "\nderived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\n\nCalling print with base-class pointer to "
      << "derived-class object\ninvokes base-class print "
      << "function on that derived-class object:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print
   cout << endl;
} // end main
```

- ❑an object of a derived class can be treated as an object of its base class.
- ❑pointer specific

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 3 of 5.)

Invoking Base-Class Functions from Derived-Class Objects (cont.)

- Assign the address of derived-class object to base-class pointer,
  - invoke member function `print` from base class.
  - This "crossover" is allowed because an object of a derived class *is an* object of its base class.
- The output of each `print` member-function invocation in this program reveals
  - the invoked functionality depends on the type of the handle (i.e., the pointer or reference type) used to invoke the function, not the type of the object to which the handle points.

Aiming Derived-Class Pointers at Base-Class Objects

- We aim a derived-class pointer at a base-class object.
  - Assign the address of base-class object to derived-class pointer
  - C++ compiler generates an error.
  - The compiler prevents this assignment, because a `CommissionEmployee` (base-class object) is *not* a `BasePlusCommissionEmployee`. (derived-class object)

```cpp
// Fig. 13.2: fig13_02.cpp
// Aiming a derived-class pointer at a base-class object.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // aim derived-class pointer at base-class object
   // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
   basePlusCommissionEmployeePtr = &commissionEmployee;
} // end main
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp8_examples\ch13\Fig13_02\fig13_02.cpp(14) : error C2440: '=' :
   cannot convert from 'CommissionEmployee *' to 'BasePlusCommissionEmployee
*'
        Cast from base to derived requires dynamic_cast or static_cast
```

Fig. 13.2 | Aiming a derived-class pointer at a base-class object.

Derived-Class Member-Function Calls via Base-Class Pointers

- Off a base-class pointer, the compiler allows us to invoke *only* base-class member functions.
- If a base-class pointer is aimed at a derived-class object, and
  - an attempt is made to access a *derived-class-only member function*,
  - a compilation error will occur.
- Shows the consequences of attempting to invoke a derived-class member function off a base-class pointer.

```
// Fig. 13.3: fig13_03  .cpp
// Attempting to invoke derived-class-only member functions
// through a base-class pointer.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

```

Fig. 13.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 1 of 3.)

```cpp
int main()
{
   CommissionEmployee *commissionEmployeePtr = 0; // base class
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class

   // aim base-class pointer at derived-class object
   commissionEmployeePtr = &basePlusCommissionEmployee;

   // invoke base-class member functions on derived-class
   // object through base-class pointer (allowed)
   string firstName = commissionEmployeePtr->getFirstName();
   string lastName = commissionEmployeePtr->getLastName();
   string ssn = commissionEmployeePtr->getSocialSecurityNumber();
   double grossSales = commissionEmployeePtr->getGrossSales();
   double commissionRate = commissionEmployeePtr->getCommissionRate();

   // attempt to invoke derived-class-only member functions
   // on derived-class object through base-class pointer (disallowed)
   double baseSalary = commissionEmployeePtr->getBaseSalary();
   commissionEmployeePtr->setBaseSalary( 500 );
} // end main
```

Fig. 13.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 3.)

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp8_examples\ch13\Fig13_03\fig13_03.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp8_examples\ch13\Fig13_03\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\cpphtp8_examples\ch13\Fig13_03\fig13_03.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp8_examples\ch13\Fig13_03\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

GNU C++ compiler error messages:

```
fig13_03.cpp:26: error: 'getBaseSalary' undeclared (first use this function)
fig13_03.cpp:27: error: 'setBaseSalary' undeclared (first use this function)
```

Fig. 13.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 3 of 3.)

Derived-Class Member-Function Calls via Base-Class Pointers (cont.)

- The compiler will allow access to derived-class-only members from a base-class pointer that is aimed at a derived-class object *if* we explicitly cast the base-class pointer to a derived-class pointer
  - known as downcasting.
- Downcasting allows a derived-class-specific operation on a derived-class object pointed to by a base-class pointer.
- After a downcast, the program *can* invoke derived-class functions *that are not in the base class*.

Virtual Functions

- Why `virtual` functions are useful?
- Consider a base class `Shape`.
  - classes such as `Circle`, `Triangle`, `Rectangle` and `Square` are all derived from base class `Shape`.
  - Each of these classes might be endowed with the ability to draw itself via a member function `draw`.
  - Although each class has its own `draw` function, the function for each shape is quite different.
  - In a program that draws a set of shapes, it would be useful to be able to treat all the shapes generically as objects of the base class `Shape`.
  - To draw any shape,
    - simply use a base-class `Shape` pointer to invoke function `draw`
    - let the program determine dynamically (i.e., at runtime) which derived-class `draw` function to use
      - based on the type of the object to which the base-class `Shape` pointer points at any given time.

Virtual Functions (cont.)

- To enable this behavior, declare `draw` in the base class as a `virtual` function
  - override `draw` in each of the derived classes to draw the appropriate shape.
- From an implementation perspective, *overriding* a function is no different than redefining one.
  - An overridden function in a derived class has the *same signature and return type* (i.e., *prototype*) as the function it overrides in its base class.
- If we declare the base-class function as `virtual`, we can override that function to enable polymorphic behavior.
- We declare a `virtual` function by preceding the function’s prototype with the key-word `virtual` in the base class.

Virtual Functions (cont.)

- Invokes a `virtual` function through
  - a base-class pointer to a derived-class object (e.g., `shapePtr->draw()`)
  - a base-class reference to a derived-class object (e.g., `shapeRef.draw()`)
  - the program will choose the correct derived-class function dynamically (i.e., at execution time) *based on the object type—not the pointer or reference type*.
  - Known as dynamic binding or late binding.
- A `virtual` function is called by referencing a specific object by name and using the dot member-selection operator (e.g., `squareObject.draw()`),
  - the function invocation is resolved at compile time (this is called static binding)
  - the `virtual` function that is called is the one defined for (or inherited by) the class of that particular object
    - this is not polymorphic behavior.
- Dynamic binding with `virtual` functions occurs only off pointer handles.

Observations: Virtual Functions

- With virtual functions, the type of the object determines which version of a virtual function to invoke
  - not the type of the handle (pointer or reference) used to invoke the member functions
- When a derived class chooses not to override a virtual function from its base class, the derived class simply inherits its base class virtual functions implementation.

Virtual Functions (cont.)

- classes `CommissionEmployee` and `BasePlusCommissionEmployee`
- The only new feature in these files is that we specify each class's `earnings` and `print` member functions as `virtual`
- Functions `earnings` and `print` are `virtual` in class `CommissionEmployee`,
  - class `BasePlusCommissionEmployee`'s `earnings` and `print` functions override class `CommissionEmployee`'s.
- Now, if we aim a base-class `CommissionEmployee` pointer at a derived-class `BasePlusCommissionEmployee` object
  - the `BasePlusCommissionEmployee` object's corresponding function will be invoked.

```
// Fig. 13.4: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using namespace std;

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

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

```

Fig. 13.4 | CommissionEmployee class header declares earnings and print as virtual. (Part 1 of 2.)

```
   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif
```

Fig. 13.4 | `CommissionEmployee` class header declares `earnings` and `print` as `virtual`. (Part 2 of 2.)

```
// Fig. 13.5: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

```

Fig. 13.5 | BasePlusCommissionEmployee class header declares earnings and print functions as virtual. (Part 1 of 2.)

```
class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Fig. 13.5 | BasePlusCommissionEmployee class header declares earnings and print functions as virtual. (Part 2 of 2.)

Virtual Functions (cont.)

- Declaring a member function `virtual` causes the program to dynamically determine which function to invoke
  - based on the type of object to which the handle points, rather than on the type of the handle.

Virtual Functions (cont.)

```
// Fig. 13.6: fig13_06.cpp
// Introducing polymorphism, virtual functions and dynamic binding.
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 1 of 6.)

Static Binding

```
    // create derived-class pointer
    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

    // set floating-point output formatting
    cout << fixed << setprecision( 2 );

    // output objects using static binding
    cout << "Invoking print function on base-class and derived-class "
       << "\nobjects with static binding\n\n";
    commissionEmployee.print(); // static binding
    cout << "\n\n";
    basePlusCommissionEmployee.print(); // static binding

    // output objects using dynamic binding
    cout << "\n\n\nInvoking print function on base-class and "
       << "derived-class \nobjects with dynamic binding";

```

Static binding

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 2 of 6.)

```cpp
   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto base-class object invokes base-class "
      << "print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with derived-class "
      << "pointer\nto derived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto derived-class object invokes derived-class "
      << "print function:\n\n";

```

Base-class pointer to derived-class object

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 3 of 6.)

```
    // polymorphism; invokes BasePlusCommissionEmployee's print;
    // base-class pointer to derived-class object
    commissionEmployeePtr->print();
    cout << endl;
} // end main
```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 4 of 6.)

- *Virtual declaration makes it object specific, not pointer specific.*
- *Now print() from derived-class is called instead of base-class*

Abstract Classes and pure virtual Functions

- There are cases in which it's useful to define *classes from which you never intend to instantiate any objects*.
  - Such classes are called abstract classes.
  - These classes normally are used as base classes in inheritance hierarchies
- These classes cannot be used to instantiate objects, because, abstract classes are *incomplete*
  - derived classes must define the "missing pieces."
- An abstract class provides a base class from which other classes can inherit.
- Classes that can be used to instantiate objects are called concrete classes.
  - Such classes define every member function they declare.

Abstract Classes and pure virtual Functions (cont.)

- Abstract base classes are *too generic* to define real objects;
  - we need to be *more specific* before we can think of instantiating objects.
- For example, if someone tells you to “draw the two-dimensional shape,” what shape would you draw?
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- An inheritance hierarchy does not need to contain any abstract classes, but many object-oriented systems have class hierarchies headed by abstract base classes.
  - In some cases, abstract classes constitute the top few levels of the hierarchy.

Abstract Classes and pure virtual Functions (cont.)

- A good example of this is the shape hierarchy, which begins with abstract base class `Shape`.
- A class is made abstract by declaring one or more of its `virtual` functions to be "pure."
  - A pure `virtual` function is specified by placing "= `0`" in its declaration, as in

    ```
    virtual void draw() const = 0; // pure virtual
       function
    ```
- The "= `0`" is a pure specifier.
- Pure `virtual` functions *do not provide implementations*.

Abstract Classes and pure virtual Functions (cont.)

- Every concrete derived class *must override all* base-class pure `virtual` functions with concrete implementations of those functions.
- The difference between a `virtual` function and a pure `virtual` function is that
  - a `virtual` function has an implementation and gives the derived class the *option* of overriding the function.
  - By contrast, a pure `virtual` function does not provide an implementation and *requires* the derived class to override the function for that derived class to be concrete; otherwise the derived class remains *abstract*.
- Pure `virtual` functions are used when it does not make sense for the base class to have an implementation of a function,
  - you want all concrete derived classes to implement the function.

Abstract Classes and Pure virtual Functions (cont.)

- Although we *cannot* instantiate objects of an abstract base class
  - we *can* use the abstract base class to *declare pointers and references* that can refer to objects of any concrete classes derived from the abstract class.
- Programs typically use such pointers and references to manipulate derived-class objects *polymorphically*.

Observations

- An abstract class defines a common public interface for the various classes in a class hierarchy
  - An abstract class contains one or more pure virtual functions that concrete derived classes must override.
- Failure to override a pure virtual function in a derived class makes that class abstract
  - Attempting to instantiate an object of an abstract class causes a compilation error
- An abstract class has at least one pure virtual function
  - An abstract class also can have data members and concrete functions (including constructors and destructors) which are subject to the normal rules of inheritance by derived classes

Case Study: Payroll System Using Polymorphism (cont.)

- Problem: A company pays its employees weekly. The employees are of three types:
  - *salaried employees* are paid a fixed weekly salary regardless of the number of hours worked,
  - *commission employees* are paid a percentage of their sales and
  - *base-salary-plus-commission employees* receive a base salary plus a percentage of their sales.
  - The company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries.
  - The company wants to implement a C++ program that performs its payroll calculations polymorphically.
- We use abstract class `Employee` to represent the general concept of an employee.

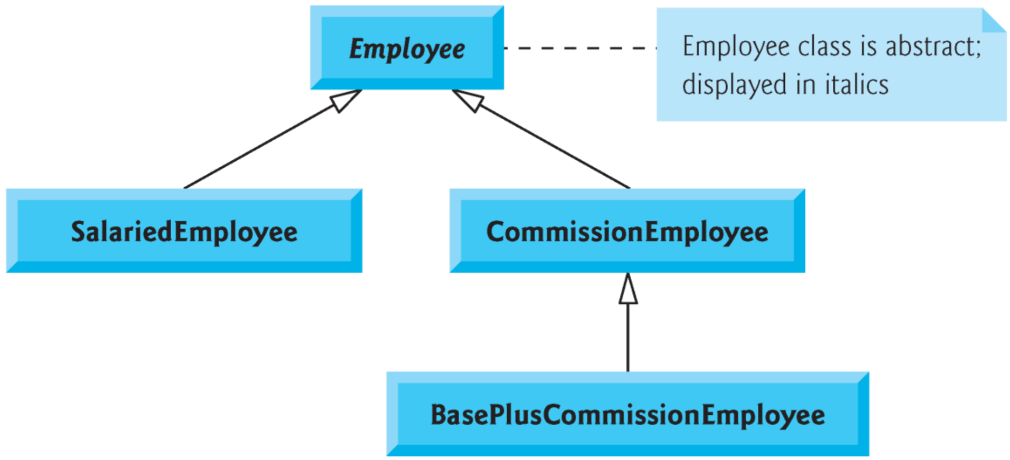


Fig. 13.7 | Employee hierarchy UML class diagram.

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	*weeklySalary*	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*
Commission-Employee	*commissionRate* * *grossSales*	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(*commissionRate* * *grossSales*) + *baseSalary*	base-salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 13.8 | Polymorphic interface for the `Employee` hierarchy classes.

Abstract Base Class: Employee

```
// Fig. 13.9: Employee.h
// Employee abstract base class.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string> // C++ standard string class
using namespace std;

```

Fig. 13.9 | Employee class header. (Part 1 of 2.)

Abstract Base Class: Employee

```
class Employee
{
public:
   Employee( const string &, const string &, const string & );

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN

   // pure virtual function makes Employee an 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
```

Fig. 13.9 | Employee class header. (Part 2 of 2.)

Abstract Base Class: Employee

```
// Fig. 13.10: Employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

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

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

```

Fig. 13.10 | Employee class implementation file. (Part 1 of 3.)

Abstract Base Class: Employee

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

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

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

```

Fig. 13.10 | Employee class implementation file. (Part 2 of 3.)

Abstract Base Class: Employee

```cpp
// set social security number
void Employee::setSocialSecurityNumber( const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

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

// print Employee's information (virtual, but not pure virtual
void Employee::print() const
{
   cout << getFirstName() << ' ' << getLastName()
      << "\nsocial security number: " << getSocialSecurityNumber();
} // end function print
```

- *Abstract Base Class Employee's print () is a virtual member function but not pure virtual function*
- *It has an implementation*
- *pure virtual function earnings() has no definition/implementation*

Fig. 13.10 | Employee class implementation file. (Part 3 of 3.)

Concrete Derived Class SalariedEmployee

```cpp
// Fig. 13.11: SalariedEmployee.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 string &,
      const string &, double = 0.0 );

   void setWeeklySalary( double ); // set weekly salary
   double getWeeklySalary() const; // return weekly salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print SalariedEmployee object
private:
   double weeklySalary; // salary per week
}; // end class SalariedEmployee

#endif // SALARIED_H
```

Class `SalariedEmployee` derives from class `Employee`.

Override Abstract Base-Class's print() & earnings() function

Fig. 13.11 | `SalariedEmployee` class header.

Concrete Derived Class SalariedEmployee (cont.)

- Function `earnings` overrides pure `virtual` function `earnings` in `Employee` to provide a *concrete* implementation that returns the `SalariedEmployee`'s weekly salary.
- If we did not implement `earnings`, class `SalariedEmployee` would be an abstract class.
- In class `SalariedEmployee`'s header, we declared member functions `earnings` and `print` as `virtual`
  - This is redundant.
  - We defined them as `virtual` in base class `Employee`, so they remain `virtual` functions throughout the class hierarchy.

Concrete Derived Class SalariedEmployee (cont.)

```
// Fig. 13.12: SalariedEmployee.cpp
// SalariedEmployee class member-function definitions.
#include <iostream>
#include "SalariedEmployee.h" // SalariedEmployee class definition
using namespace std;

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

// set salary
void SalariedEmployee::setWeeklySalary( double salary )
{
   if ( salary >= 0.0 )
      weeklySalary = salary;
   else
      throw invalid_argument( "Weekly salary must be >= 0.0" );
} // end function setWeeklySalary

```

Fig. 13.12 | SalariedEmployee class implementation file. (Part 1 of 2.)

Concrete Derived Class SalariedEmployee (cont.)

```
// return salary
double SalariedEmployee::getWeeklySalary() const
{
   return weeklySalary;
} // end function getWeeklySalary

// calculate earnings;
// override pure virtual function earnings in Employee
double SalariedEmployee::earnings() const
{
   return getWeeklySalary();
} // end function earnings

// print SalariedEmployee's information
void SalariedEmployee::print() const
{
   cout << "salaried employee: ";
   Employee::print(); // reuse abstract base-class print function
   cout << "\nweekly salary: " << getWeeklySalary();
} // end function print
```

- Override pure virtual function earnings()

- Definition of print() in derived-class

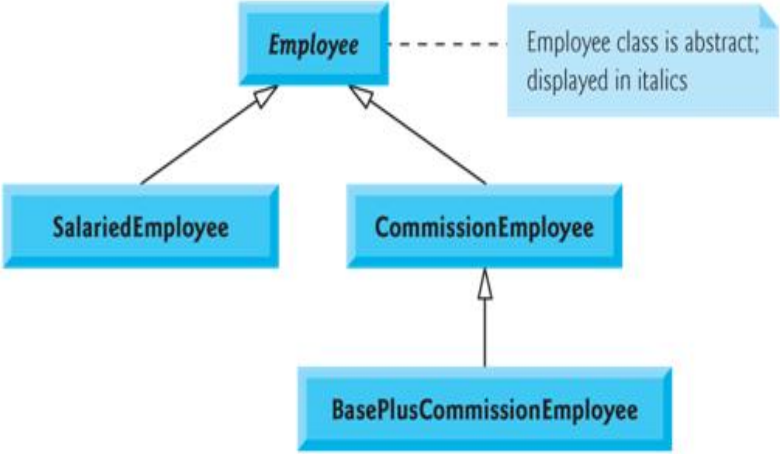
Fig. 13.12 | SalariedEmployee class implementation file. (Part 2 of 2.)

Concrete Derived Class SalariedEmployee (cont.)

- Function `print` of class `SalariedEmployee` overrides `Employee` function `print`.
- If class `SalariedEmployee` did not override `print`, `SalariedEmployee` would inherit the `Employee` version of `print`.

Another Concrete Derived Class CommissionEmployee

- Class `CommissionEmployee` derives from `Employee`.
- The constructor passes the first name, last name and social security number to the `Employee` constructor to initialize `Employee`'s `private` data members.
- Function `print` calls base-class function `print` to display the `Employee`-specific information.



Another Derived Class CommissionEmployee

```
// Fig. 13.13: CommissionEmployee.h
// CommissionEmployee class derived from Employee.
#ifndef COMMISSION_H
#define COMMISSION_H

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

```

Fig. 13.13 | CommissionEmployee class header. (Part 1 of 2.)

Another Derived Class CommissionEmployee

```
class CommissionEmployee : public Employee
{
public:
   CommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif // COMMISSION_H
```

Fig. 13.13 | CommissionEmployee class header. (Part 2 of 2.)

Another Derived Class CommissionEmployee

```
// Fig. 13.14: CommissionEmployee.cpp
// CommissionEmployee class member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee( const string &first,
   const string &last, const string &ssn, double sales, double rate )
   : Employee( first, last, ssn )
{
   setGrossSales( sales );
   setCommissionRate( rate );
} // end CommissionEmployee constructor

```

Fig. 13.14 | CommissionEmployee class implementation file. (Part 1 of 4.)

Another Derived Class CommissionEmployee

```
// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
   if ( sales >= 0.0 )
      grossSales = sales;
   else
      throw invalid_argument( "Gross sales must be >= 0.0" );
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
   return grossSales;
} // end function getGrossSales

```

Fig. 13.14 | CommissionEmployee class implementation file. (Part 2 of 4.)

Another Derived Class CommissionEmployee

```
// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   if ( rate > 0.0 && rate < 1.0 )
      commissionRate = rate;
   else
      throw invalid_argument( "Commission rate must be > 0.0 and < 1.0" );
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
   return commissionRate;
} // end function getCommissionRate

// calculate earnings; override pure virtual function earnings in Employee
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

```

- Override the pure virtual function earnings()

Fig. 13.14 | CommissionEmployee class implementation file. (Part 3 of 4.)

Another Derived Class CommissionEmployee

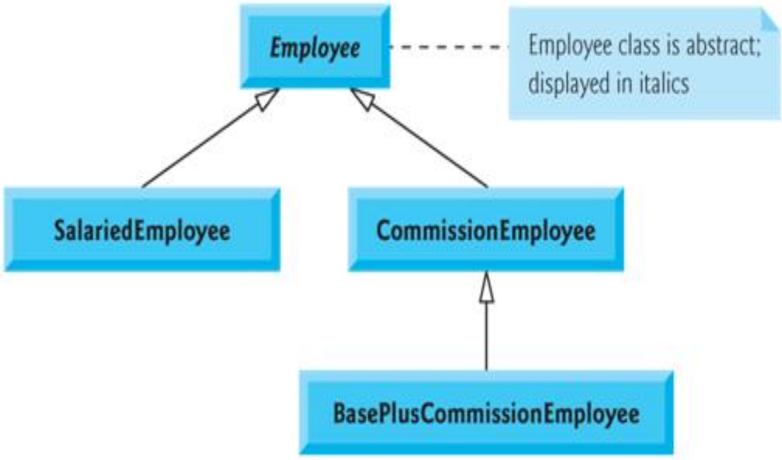
```
// print CommissionEmployee's information
void CommissionEmployee::print() const
{
   cout << "commission employee: ";
   Employee::print(); // code reuse
   cout << "\ngross sales: " << getGrossSales()
      << "; commission rate: " << getCommissionRate();
} // end function print
```

- Definition of print() in derived-class

Fig. 13.14 | CommissionEmployee class implementation file. (Part 4 of 4.)

Indirect Concrete Derived Class BasePlusCommissionEmployee

- Class `BasePlusCommissionEmployee` directly inherits from class `CommissionEmployee`
  - it is an indirect derived class of class `Employee`.
- `BasePlusCommissionEmployee`'s `print` function outputs
  - `"base-salaried"`, followed by the output of base-class `CommissionEmployee`'s `print` function (another example of code reuse), then the base salary.



Indirect Concrete Derived Class BasePlusCommissionEmployee

```cpp
// Fig. 13.15: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include "CommissionEmployee.h" // CommissionEmployee class definition

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

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary per week
}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

Fig. 13.15 | BasePlusCommissionEmployee class header.

Indirect Concrete Derived Class BasePlusCommissionEmployee

```
// Fig. 13.16: BasePlusCommissionEmployee.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

```

Fig. 13.16 | BasePlusCommissionEmployee class implementation file. (Part 1 of 3.)

Indirect Concrete Derived Class BasePlusCommissionEmployee

```
// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   if ( salary >= 0.0 )
      baseSalary = salary;
   else
      throw invalid_argument( "Salary must be >= 0.0" );
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary

// calculate earnings;
// override virtual function earnings in CommissionEmployee
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

```

Fig. 13.16 | BasePlusCommissionEmployee class imple
file. (Part 2 of 3.)

- Definition of virtual function earnings() in derived-class

```
// calculate earnings; override pure virtual function earnings in Employee
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

```

Fig. 13.14 | CommissionEmployee class implementation file. (Part 3

Indirect Concrete Derived Class BasePlusCommissionEmployee

```
// print BasePlusCommissionEmployee's information
void BasePlusCommissionEmployee::print() const
{
   cout << "base-salaried ";
   CommissionEmployee::print(); // code reuse
   cout << "; base salary: " << getBaseSalary();
} // end function print
```

Fig. 13.16 | BasePlusCommissionEmployee class implementation file. (Part 3 of 3.)

- Definition of print() in derived-class

Demonstrating Polymorphic Processing

- Create an object of each of the three concrete classes `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`.
- Manipulates these objects
  - static binding,
  - polymorphically, using a `vector` of `Employee` pointers.
- Each member-function invocation is an example of static binding
  - at compile time, because we are using name handles (not pointers or references that could be set at execution time)
  - the compiler can identify each object's type to determine which `print` and `earnings` functions are called.

Example: Polymorphic Processing

```
// Fig. 13.17: fig13_17.cpp
// Processing Employee derived-class objects individually
// and polymorphically using dynamic binding.
#include <iostream>
#include <iomanip>
#include <vector>
#include "Employee.h"
#include "SalariedEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

void virtualViaPointer( const Employee * const ); // prototype
void virtualViaReference( const Employee & ); // prototype

```

Fig. 13.17 | Employee class hierarchy driver program. (Part 1 of 7.)

*const Employee ** :pointer to an object and the object cannot be modified.
*Employee * const* :you cannot change what the pointer points to.
*const Employee * const* :a pointer which cannot be changed to point to something else, nor can it be used to change the object it points to.

Example: Polymorphic Processing

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

   // create derived-class objects
   SalariedEmployee salariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

   cout << "Employees processed individually using static binding:\n\n";

```

Fig. 13.17 | Employee class hierarchy driver program. (Part 2 of 7.)

Example: Polymorphic Processing

```
   // output each Employee's information and earnings using static binding
   salariedEmployee.print();
   cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
   commissionEmployee.print();
   cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
   basePlusCommissionEmployee.print();
   cout << "\nearned $" << basePlusCommissionEmployee.earnings()
      << "\n\n";

   // create vector of three base-class pointers
   vector < Employee * > employees( 3 );

   // initialize vector with Employees
   employees[ 0 ] = &salariedEmployee;
   employees[ 1 ] = &commissionEmployee;
   employees[ 2 ] = &basePlusCommissionEmployee;

   cout << "Employees processed polymorphically via dynamic binding:\n\n";

```

Fig. 13.17 | Employee class hierarchy driver program. (Part 3 of 7.)

Example: Polymorphic Processing

```
   // call virtualViaPointer to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class pointers:\n\n";

   for ( size_t i = 0; i < employees.size(); ++i )
      virtualViaPointer( employees[ i ] );

   // call virtualViaReference to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class references:\n\n";

   for ( size_t i = 0; i < employees.size(); ++i )
      virtualViaReference( *employees[ i ] ); // note dereferencing
} // end main

// call Employee virtual functions print and earnings off a
// base-class pointer using dynamic binding
void virtualViaPointer( const Employee * const baseClassPtr )
{
   baseClassPtr->print();
   cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
} // end function virtualViaPointer

```

Fig. 13.17 | Employee class hierarchy driver program. (Part 4 of 7.)

Example: Polymorphic Processing

```
// call Employee virtual functions print and earnings off a
// base-class reference using dynamic binding
void virtualViaReference( const Employee &baseClassRef )
{
   baseClassRef.print();
   cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
} // end function virtualViaReference
```

```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Fig. 13.17 | Employee class hierarchy driver program. (Part 5 of 7.)

Example: Polymorphic Processing

```
Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Fig. 13.17 | Employee class hierarchy driver program. (Part 6 of 7.)

Example: Polymorphic Processing

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

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Fig. 13.17 | Employee class hierarchy driver program. (Part 7 of 7.)

Polymorphic Processing (cont.)

- `vector employees`, which contains three `Employee` pointers.
- `employees[ 0 ]` at object `salariedEmployee`.
- `employees[ 1 ]` at object `commissionEmployee`.
- `employees[ 2 ]` at object `basePlusCommissionEmployee`.
- The compiler allows these assignments, because a `SalariedEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`.

Polymorphic Processing (cont.)

- Function `virtualViaPointer` receives in parameter `baseClassPtr` (of type `const Employee * const`) the address stored in an `employees` element.
- Each call to `virtualViaPointer` uses `baseClassPtr` to invoke `virtual` functions `print` and `earnings`
- Note that function `virtualViaPointer` does not contain any `SalariedEmployee`, `CommissionEmployee` or `BasePlusCommissionEmployee` type information.
- The function knows only about base-class type `Employee`.
- The output illustrates that the appropriate functions for each class are indeed invoked and that each object's proper information is displayed.

Polymorphic Processing (cont.)

- Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee &`) a reference to the object obtained by dereferencing the pointer stored in each `employees` element.
- Each call to `virtualViaReference` invokes `virtual` functions `print` and `earnings` via reference `baseClassRef` to demonstrate that polymorphic processing occurs with base-class references as well.
- Each `virtual`-function invocation calls the function on the object to which `baseClassRef` refers at runtime.
- This is another example of dynamic binding.
- The output produced using *base-class references* is identical to the output produced using *base-class pointers*.

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

- Internal implementation of polymorphism, `virtual` functions and dynamic binding.
- More importantly, it will help you appreciate the overhead of polymorphism
  - in terms of additional memory consumption and processor time.
- Polymorphism is accomplished through three levels of pointers (i.e., "triple indirection").
- How an executing program uses these data structures to execute `virtual` functions and achieve the dynamic binding associated with polymorphism.

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- C++ compiles a class that has one or more `virtual` functions
  - builds a virtual function table (*vtable*) for that class.
- An executing program uses the *vtable* to select the proper function implementation each time a `virtual` function of that class is called.
  - the *vtables* for classes `Employee`, `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`.
- In the *vtable* for class `Employee`, the first function pointer is set to 0 (i.e., the null pointer).
  - This is done because function `earnings` is a pure `virtual` function and therefore lacks an implementation.

Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood” (cont.)

- The second function pointer points to function `print`, which displays the employee’s full name and social security number.
- Any class that has one or more null pointers in its *vtable* is an *abstract* class.
- Classes without any null *vtable* pointers are *concrete* classes.
- Class `SalariedEmployee` overrides function `earnings` to return the employee’s weekly salary,
  - the function pointer points to the `earnings` function of class `SalariedEmployee`.
- `SalariedEmployee` also overrides `print`, so the corresponding function pointer points to the `SalariedEmployee` member function that prints `"salaried employee: "` followed by the employee’s name, social security number and weekly salary.

Virtual function working mechanism

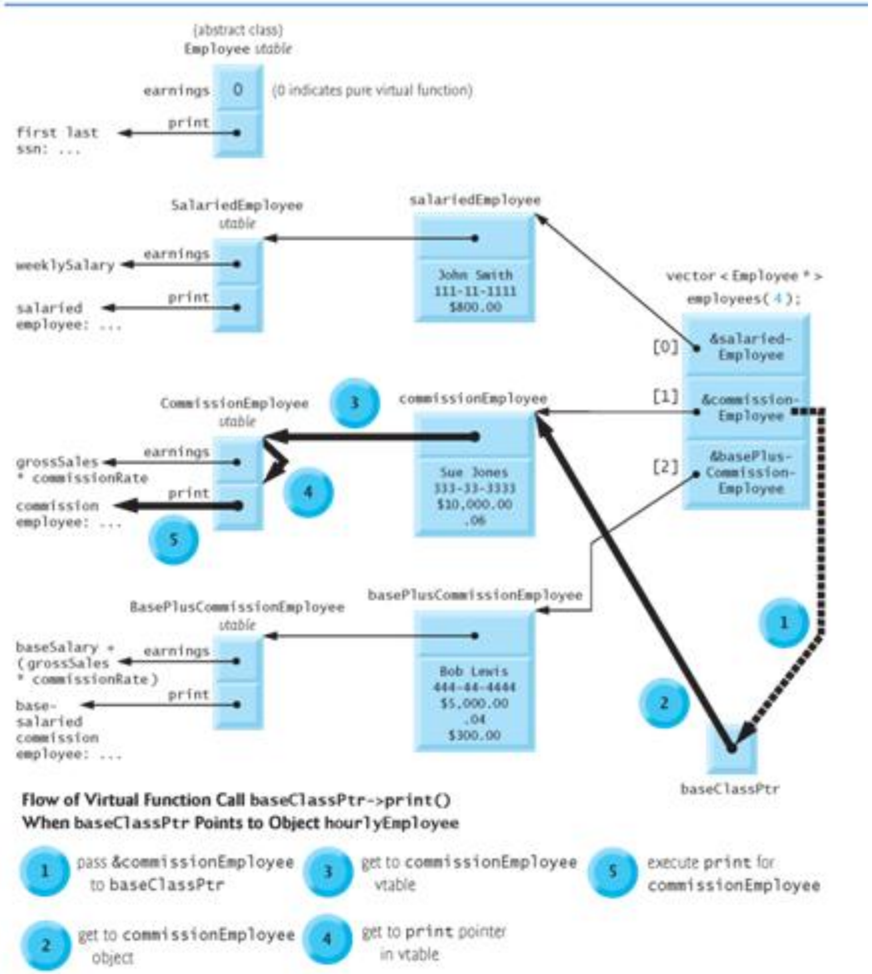


Fig. 13.18 | How virtual function calls work.

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- The `earnings` function pointer in the *vtable* for class `CommissionEmployee`
  - points to `CommissionEmployee`'s `earnings` function
  - returns the employee's gross sales multiplied by the commission rate.
- The `print` function pointer points to the `CommissionEmployee` version of the function,
  - prints the employee's type, name, social security number, commission rate and gross sales.
- As in class `SalariedEmployee`, both functions override the functions in class `Employee`.

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- The `earnings` function pointer in the *vtable* for class `BasePlusCommissionEmployee`
  - points to the `BasePlusCommissionEmployee`'s `earnings` function
  - returns the employee's base salary plus gross sales multiplied by commission rate.
- The `print` function pointer points to the `BasePlusCommissionEmployee` version of the function,
  - prints the employee's base salary plus the type, name, social security number, commission rate and gross sales.
- Both functions override the functions in class `CommissionEmployee`.

Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood” (cont.)

- Polymorphism is accomplished through an elegant data structure involving *three levels of pointers*.
- One level—the function pointers in the *vtable*.
  - These point to the actual functions that execute when a `virtual` function is invoked.
- Second level of pointers.
  - Whenever *an object of a class with one or more `virtual` functions is instantiated*, *the compiler attaches to the object a pointer to the vtable for that class*.
  - Display each of the object’s data member values.

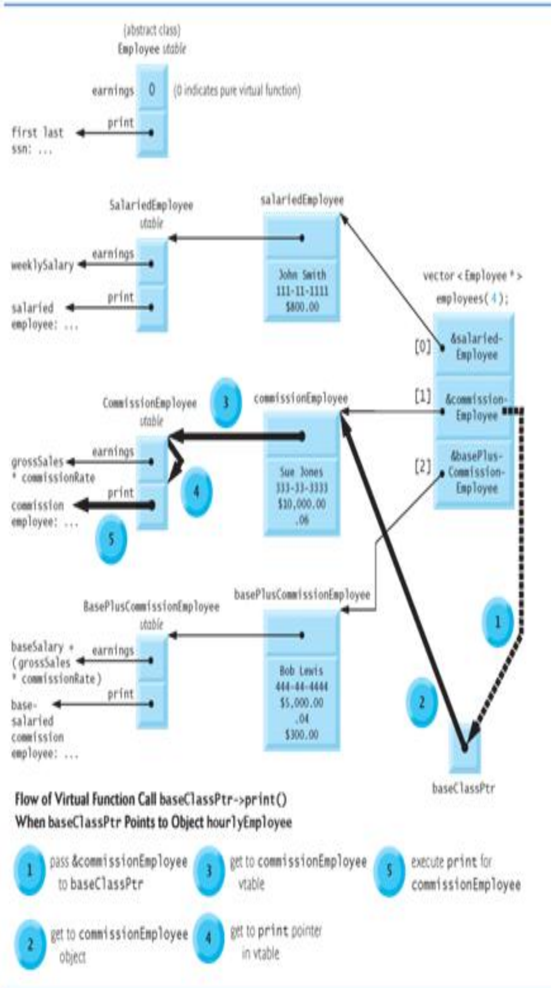


Fig. 13.18 | How `virtual` function calls work.

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- The *third level of pointers* simply contains the handles to the objects that receive the `virtual` function calls.
- The handles in this level may also be references.

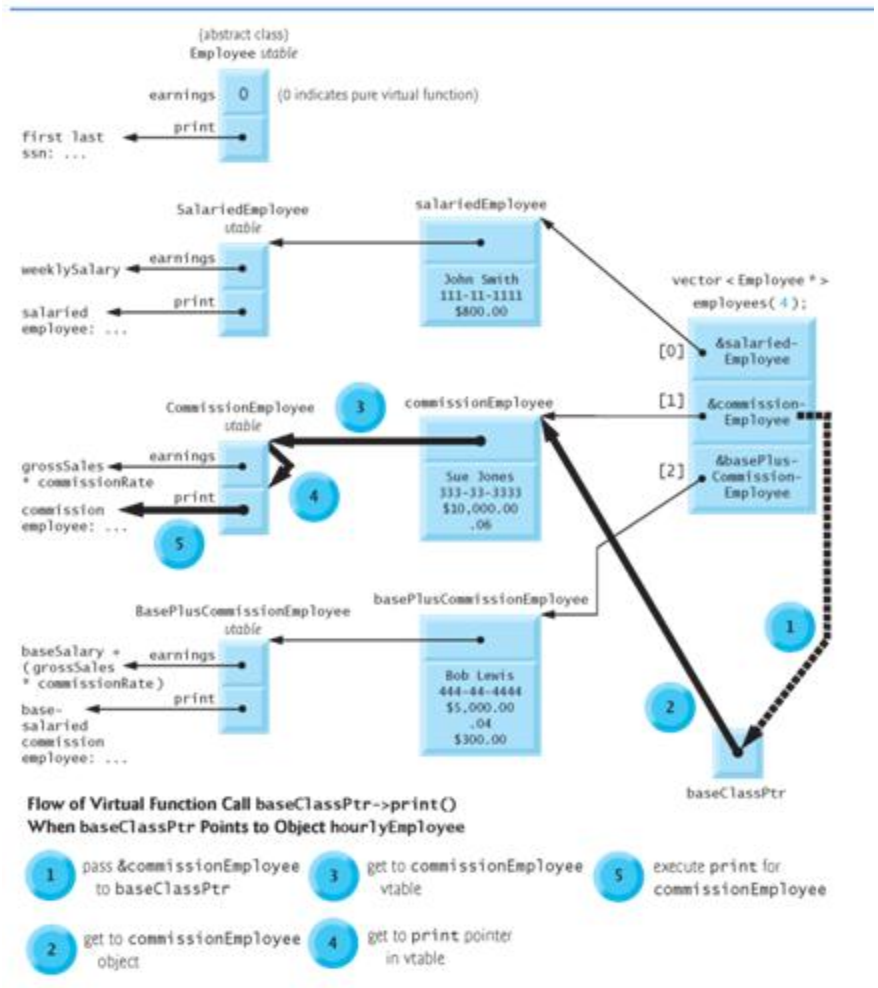


Fig. 13.18 | How `virtual` function calls work.

Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood” (cont.)

- Let’s see how a typical `virtual` function call executes.
- `vector employees` contains `Employee` pointers.
- Consider the call `baseClassPtr->print()` in function `virtualViaPointer`.
- Assume that `baseClassPtr` contains `employees[ 1 ]` (i.e., the address of object `commissionEmployee` in `employees`).
- When the compiler compiles this statement, it determines that the call is *indeed being made via a base-class pointer and that* ***`print`*** *is a* ***`virtual`*** *function.*
  - The compiler determines that `print` is the *second* entry in each of the *vtables.*
  - To locate this entry, the compiler notes that it will need to skip the first entry.

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- The compiler compiles an offset or displacement of four bytes (four bytes for each pointer on today's popular 32-bit machines, and only one pointer needs to be skipped) into the table of machine-language object-code pointers to find the code
  - that will execute the `virtual` function call.

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood" (cont.)

- The compiler generates code that performs the following operations.
  - Select the i^{th} entry of `employees`, and pass it as an argument to function `virtualViaPointer`. This sets parameter `baseClassPtr` to point to `commissionEmployee`.
  - Dereference that pointer to get to the `commissionEmployee` object.
  - Dereference `commissionEmployee`'s *vtable* pointer to get to the `CommissionEmployee` *vtable*.
  - Skip the offset of four bytes to select the `print` function pointer.
  - Dereference the `print` function pointer to form the "name" of the actual function to execute, and use the function call operator `()` to execute the appropriate `print` function.

Observations

- Polymorphism is typically implemented with virtual functions and dynamic binding in C++, is efficient.
  - We can use those capabilities with nominal impact on performance.
  - Polymorphism’s overhead is acceptable for most applications.
  - Polymorphism’s overhead may be too high for real time applications with stringent performance.

Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info

- Consider the company has decided to reward `BasePlusCommissionEmployee`s by adding 10 percent to their base salaries.
- When processing `Employee` objects polymorphically, we did not need to worry about the "specifics."
- To adjust the base salaries of `BasePlusCommissionEmployee`s, *we have to determine the specific type of each `Employee` object at execution time*, then act appropriately.
- Demonstrate the powerful capabilities of runtime type information (RTTI) and dynamic casting,
  - enable a program to determine the type of an object at execution time and act on that object accordingly.

Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info (cont.)

- Some compilers require that RTTI be enabled before it can be used in a program.
  - In Visual C++ 2010, this option is enabled by default.
- *Exercise:* Increase by 10 percent the base salary of each `BasePlusCommissionEmployee`.

Example: Downcasting

```
// Fig. 13.19: fig13_19.cpp
// Demonstrating downcasting and runtime type information.
// NOTE: You may need to enable RTTI on your compiler
// before you can execute this application.
#include <iostream>
#include <iomanip>
#include <vector>
#include <typeinfo>
#include "Employee.h"
#include "SalariedEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 1 of 5.)

Example: Downcasting

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

   // create vector of three base-class pointers
   vector < Employee * > employees( 3 );

   // initialize vector with various kinds of Employees
   employees[ 0 ] = new SalariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   employees[ 1 ] = new CommissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   employees[ 2 ] = new BasePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 2 of 5.)

Example: Downcasting

```
    // polymorphically process each element in vector employees
    for ( size_t i = 0; i < employees.size(); ++i )
    {
       employees[ i ]->print(); // output employee information
       cout << endl;

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

       // determine whether element points to base-salaried
       // commission employee
       if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
       {
          double oldBaseSalary = derivedPtr->getBaseSalary();
          cout << "old base salary: $" << oldBaseSalary << endl;
          derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
          cout << "new base salary with 10% increase is: $"
             << derivedPtr->getBaseSalary() << endl;
       } // end if

       cout << "earned $" << employees[ i ]->earnings() << "\n\n";
    } // end for
```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 3 of 5.)

Example: Downcasting

```

   // release objects pointed to by vector's elements
   for ( size_t j = 0; j < employees.size(); ++j )
   {
      // output class name
      cout << "deleting object of "
         << typeid( *employees[ j ] ).name() << endl;

      delete employees[ j ];
   } // end for
} // end main
```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 4 of 5.)

Example: Downcasting

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 5 of 5.)

Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info (cont.)

- Since we process the employees polymorphically, we cannot be certain as to which type of `Employee` is being manipulated at any given time.
- `BasePlusCommissionEmployee` employees *must* be identified when we encounter them so they can receive the 10 percent salary increase.
- To accomplish this, we use operator `dynamic_cast` to determine whether the type of each object is `BasePlusCommissionEmployee`.
  - This is the *downcast* operation.
  - Dynamically downcast `employees[i]` from type `Employee *` to type `BasePlusCommissionEmployee *`.

Using Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info (cont.)

- If the `vector` element points to an object that *is a* `BasePlusCommissionEmployee` object,
  - then that object's address is assigned to `derivedPtr`
  - otherwise, `0` is assigned to derived-class pointer `derivedPtr`.
- If the value returned by the `dynamic_cast` operator *is not* `0`
  - the object is the correct type, and
  - the `if` statement performs the special processing required for the `BasePlusCommissionEmployee` object.

Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info (cont.)

- Operator `typeid` returns a reference to an object of class `type_info`
  - contains the information about the type of its operand, including the name of that type.
- When invoked, `type_info` member function `name` returns a pointer-based string that contains the type name (e.g., `"class BasePlusCommissionEmployee"`) of the argument passed to `typeid`.
- To use `typeid`, the program must include header `<typeinfo>`

```

   // release objects pointed to by vector's elements
   for ( size_t j = 0; j < employees.size(); ++j )
   {
      // output class name
      cout << "deleting object of "
         << typeid( *employees[ j ] ).name() << endl;

      delete employees[ j ];
   } // end for
} // end main
```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 4 of 5.)

Virtual Destructors

- A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy.
- So far we have seen non virtual destructors
  - destructors that are not declared with keyword `virtual`.
- If a derived-class object with a nonvirtual destructor is destroyed explicitly by applying the `delete` operator to a base-class pointer to the object,
  - the C++ standard specifies that the behavior is undefined.
- The simple solution to this problem is to create a `virtual` destructor in the base class.
  - This makes all derived-class destructors `virtual` *even though they do not have the same name as the base-class destructor*.
- Now, if an object in the hierarchy is destroyed explicitly by applying the `delete` operator to a base-class pointer,
  - the destructor for the appropriate class is called based on the object to which the base-class pointer points.

Observations

- If a class has a virtual function; provide a virtual destructor, even if one is not required for the class.
  - ensure that a custom derived-class destructor will be invoked (if there is one) when a derived-class object is deleted via a base class pointer
- Constructor cannot be virtual
  - Declaring a constructor virtual is a compilation error.