Object-Oriented Programming
(in C++)

Polymorphism

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Important OO Concepts: P.I.E.

- Encapsulation (abstract data type)
  - "Black box" – information hiding

- Inheritance
  - Related classes share implementation and/or interface, allowing reuse of codes
- Polymorphism
  - Ability to use a class without knowing its type

Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

What Is Polymorphism? Learning by Examples

- Suppose we design a video game that manipulates objects of different classes: `Martian`, `Venutian`, and `Plutonian`
  - Each of these classes inherits from the common base class `Alien`, which contains a member function `draw()`
  - Each derived class implements this function in its own way

Video Game Example: Class Definitions

```
class Alien {
public:
  void draw() {

  }
};
```

```
class Martian
  : public Alien {
private:
  ...
};
void Martian::draw() {

}
```

Redefine draw()

```
class Venutian
  : public Alien {
private:
  ...
};
void Venutian::draw() {

}
```

Redefine draw()

```
class Plutonian
  : public Alien {
private:
  ...
};
void Plutonian::draw() {

}
```

Redefine draw()

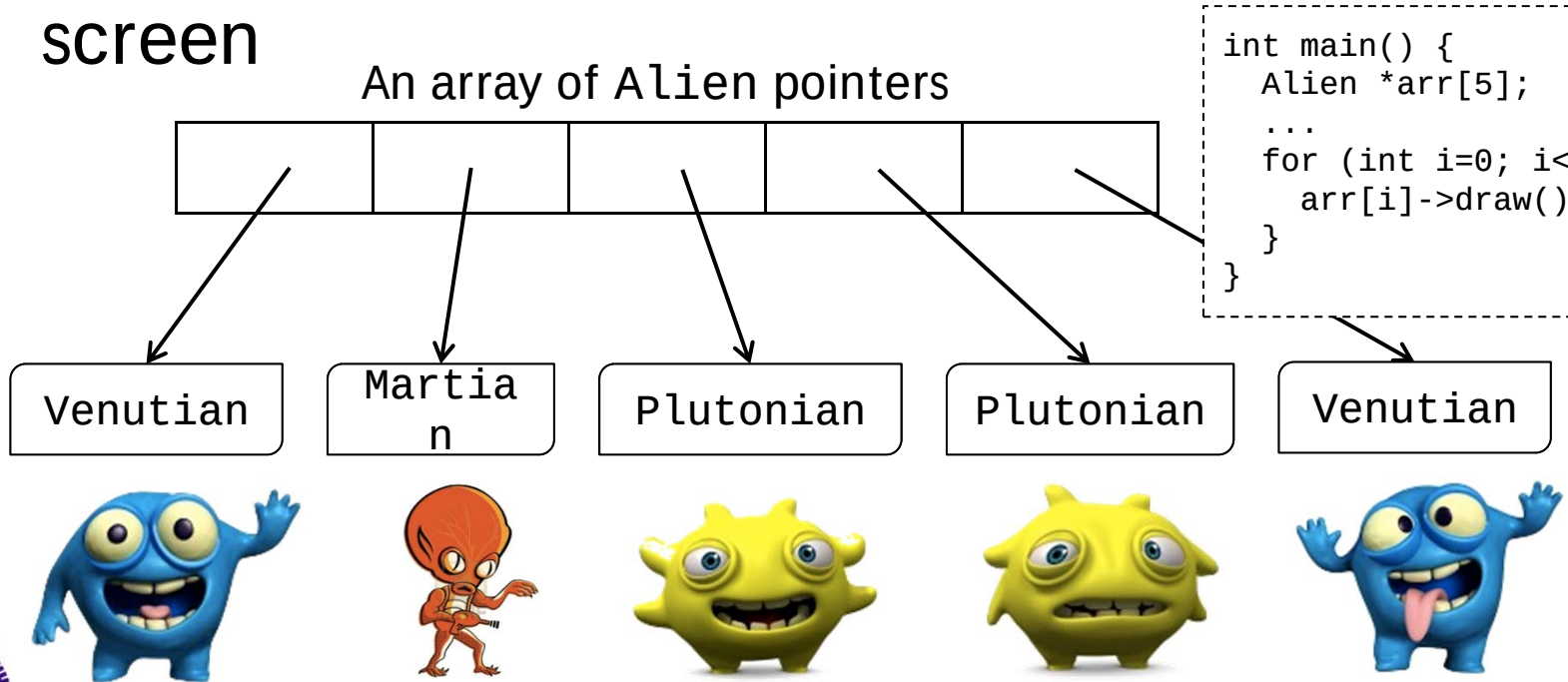
Recall: Inheritance Relationship

- Key concept
  - An object of a derived class can be treated as an object of its base class if `public` inheritance is used
  - "is-a" relationship: A object *is a* B object
    - Class A inherits from class B

- A `Martian` object is an `Alien` object
- A `Venutian` object is an `Alien` object
- A `Plutonian` object is an `Alien` object

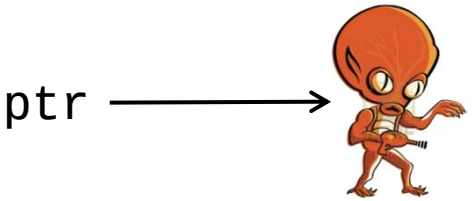
Video Game Example: Designing Screen Manager

- A screen-manager program maintains a container (e.g., an array) that holds `Alien` pointers to objects of the various classes
- The screen manager periodically sends each object the same message (`draw`) to refresh the screen



Which Function Is Invoked with Pointers?

- Suppose we have a pointer (`ptr`) to a `Martian` object, does `ptr->draw()` mean calling `Martian::draw()`?



  - In C++, invoked functionality depends on the type of the handle (i.e., the type of the pointer)
    - If the pointer is a `Alien` pointer,

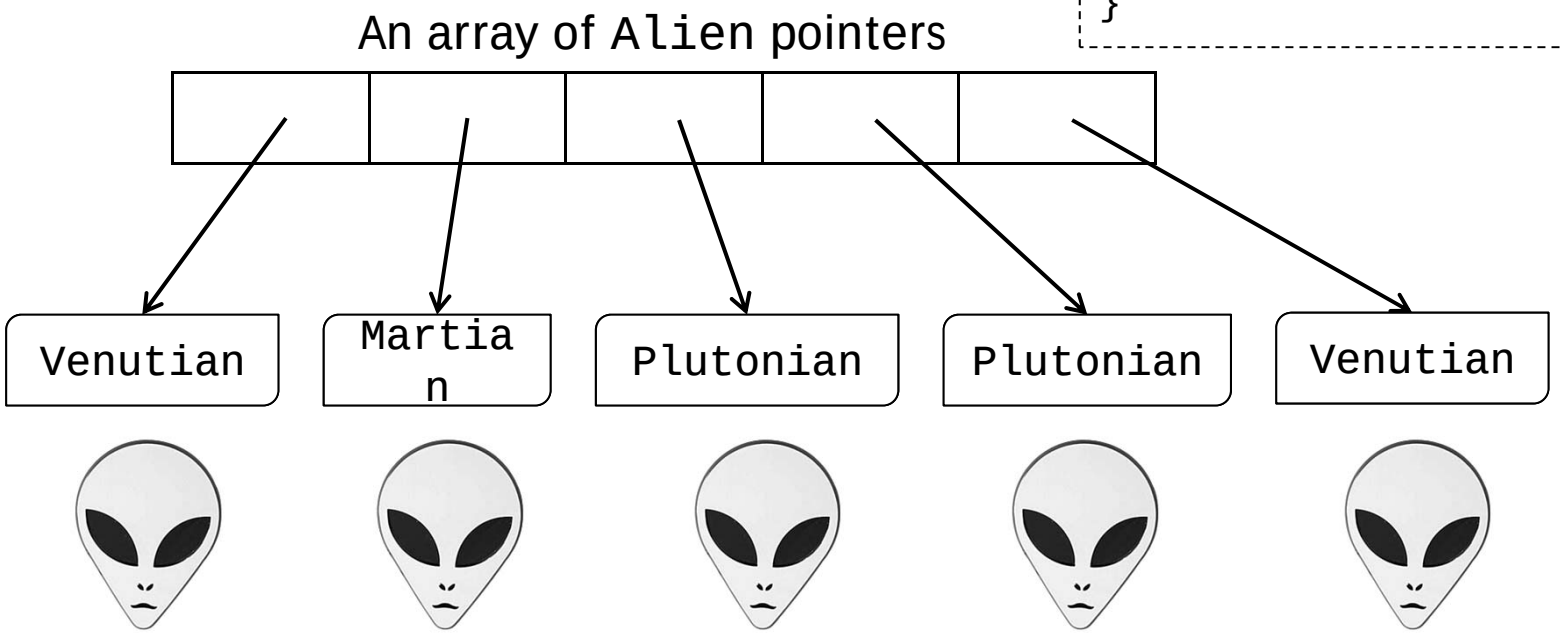
      calling `ptr->draw()` invokes `Alien::draw()`
    - If the pointer is a `Martian` pointer,

      calling `ptr->draw()` invokes `Martian::draw()`
    - If the pointer is a `Venutian` pointer,

      the pointer cannot point to the `Martian` object unless a conversion constructor/operator for converting a `Martian` object to a `Venutian` object exists

Video Game Example: Unwanted Results

```
int main() {
  Alien* arr[5];
  ...
  for (int i=0; i<5; i++) {
    arr[i]->draw();
  }
}
```



- If the pointer is a `Alien` pointer, calling `ptr->draw()` invokes `Alien::draw()`

Video Game Example: Tedious Approach

```
enum AlienType { MARTIAN, VENUTIAN, PLUTONIAN };
class Alien {
public:
  enum AlienType type;
  void draw() {
    ...
  }
};
```

```
int main() {
  Alien* arr[5];
  ...
  for (int i=0; i<5; i++) {
    switch(arr[i]->type) {
      case MARTIAN:
        Martian *martianPtr = dynamic_cast<Martian *>(arr[i]);
        martianPtr->draw();
        break;
      case VENUTIAN:
        Venutian *venutianPtr= dynamic_cast<Venutian *>(arr[i]);
        venutianPtr->draw();
        break;
      case PLUTONIAN:
        Plutonian *plutonianPtr= dynamic_cast<Plutonian *>(arr[i]);
        plutonianPtr->draw();
        break;
    }
  }
}
```

Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Video Game Example: Polymorphic Approach

```
class Alien {
public:
  virtual void draw() {


  }
};
```

```
class Martian
  : public Alien {
private:
  ...
};
void Martian::draw() {



}
```

Function overriding

```
class Venutian
  : public Alien {
private:
  ...
};
void Venutian::draw() {



}
```

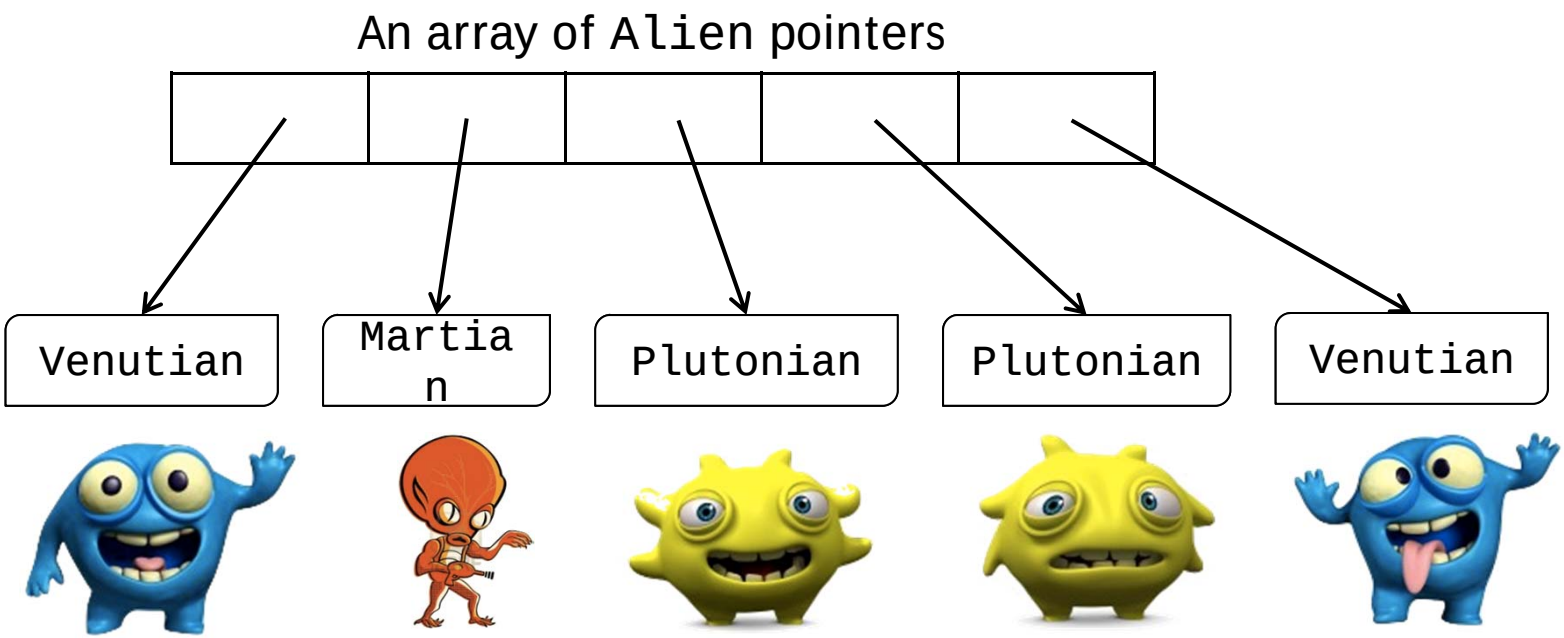
Function overriding

```
class Plutonian
  : public Alien {
private:
  ...
};
void Plutonian::draw() {



}
```

Function overriding

A Polymorphic Screen Manager

```
int main() {
  Alien *arr[5];
  ...
  for (int i=0; i<5; i++) {
    arr[i]->draw();
  }
}
```



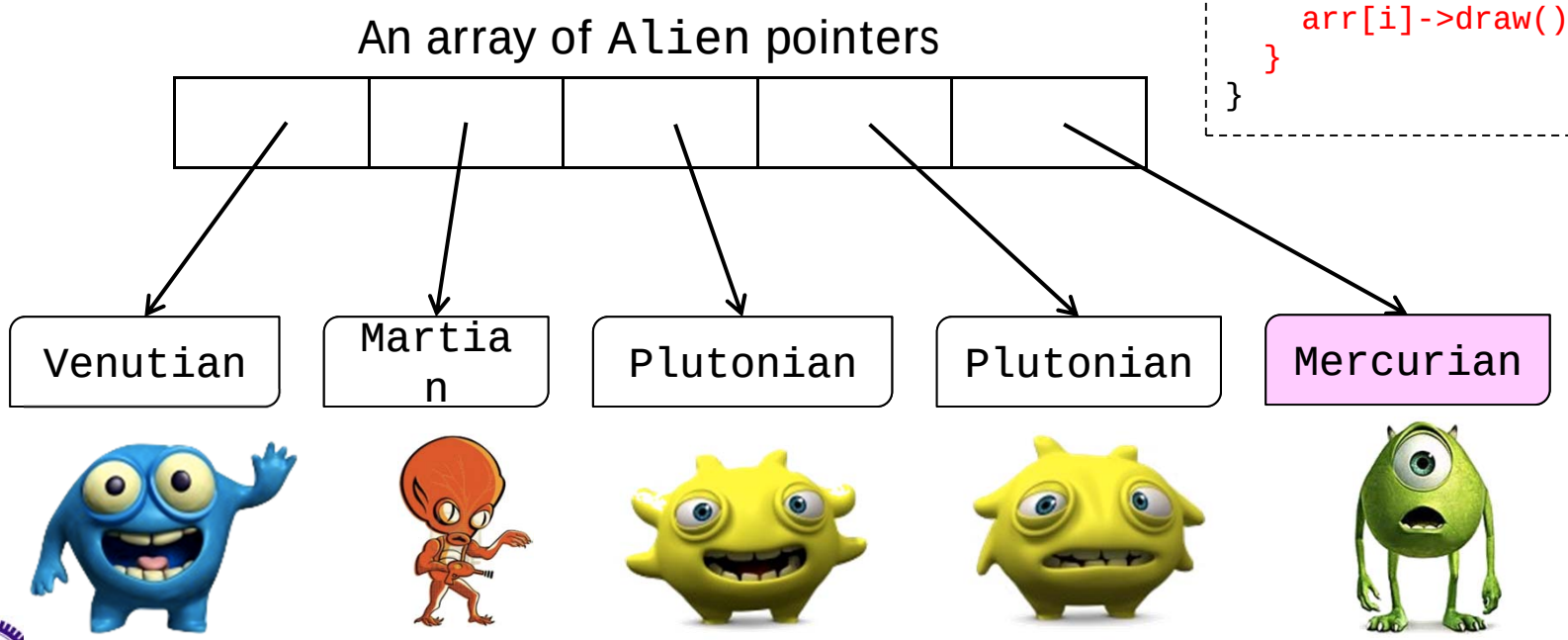
What Is Polymorphism?

- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism
- 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, as long as the new classes are part of the inheritance hierarchy that the program processes generically

A Polymorphic Screen Manager

- We don't need to modify the code for the screen manager even if a new class in the inheritance hierarchy is created

```
int main() {
  Alien *arr[5];
  ...
  for (int i=0; i<5; i++) {
    arr[i]->draw();
  }
}
```



`virtual` Functions (1/3)

- Which class's function to invoke
  - Normally
    - A handle determines which class's functionality to invoke
  - With `virtual` functions
    - **Type of the object being pointed to**, not type of the handle, determines which version of a `virtual` function to invoke
    - Allows programs to dynamically (at runtime rather than compile time) determine which function to use
      - Called dynamic binding (or late binding)

`virtual` Functions (2/3)

- Declared by preceding the function's prototype with the keyword `virtual` in base class
- Derived classes override function as appropriate
- 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
  - virtual function (base class) -> overridden function (derived class)
  - non-virtual function (base class) -> redefined function (derived class)

`virtual` Functions (3/3)

- A `virtual` function can be either overridden or not
  - A `virtual` function is not overridden
    - Inherits from its base class
  - A `virtual` function is overridden
    - Once `virtual`, always `virtual` (even not explicitly declared)
    - Good programming practice:
      - Explicitly declare these functions `virtual` at every level of the hierarchy to promote program clarity

Static Binding vs Dynamic Binding

- Static binding (not polymorphic behavior)
  - When calling a `virtual`/non-`virtual` function using a specific object with the dot operator, rather than a pointer or a reference
  - Function invocation is resolved at compile time

- Dynamic binding (polymorphic behavior)
  - Dynamic binding occurs only off pointer or reference handles when invoking `virtual` functions
  - Choosing the appropriate function to call at execution time

Video Game Example: Another Approach

If a class contains at least a pure virtual function, we cannot create an object from the abstract class

In many cases, implementation of functions in the base class makes no sense at all
We call such functions **pure virtual functions**

```
class Alien {
public:
  virtual void draw() = 0;
};
```

```
class Martian
  : public Alien {
private:
  ...
};
void Martian::draw() {

}
```

```
class Venutian
  : public Alien {
private:
  ...
};
void Venutian::draw() {

}
```

```
class Plutonian
  : public Alien {
private:
  ...
};
void Plutonian::draw() {

}
```

Pure `virtual` Functions

- Placing "`= 0`" in a virtual function's declaration
  - Example: (.h file)
    - `virtual void draw() const = 0;`
  - "`= 0`" is known as a pure specifier
  - Do not provide implementations
- Used when it does not make sense for base class to have an implementation of a function, but the programmer wants all its derived classes to implement the function

Pure `virtual` Functions vs `virtual` Functions

- A `virtual` function has an implementation and gives the derived class the *option* of overriding the function
- A pure `virtual` function does not provide an implementation and *requires* the derived class to override the function

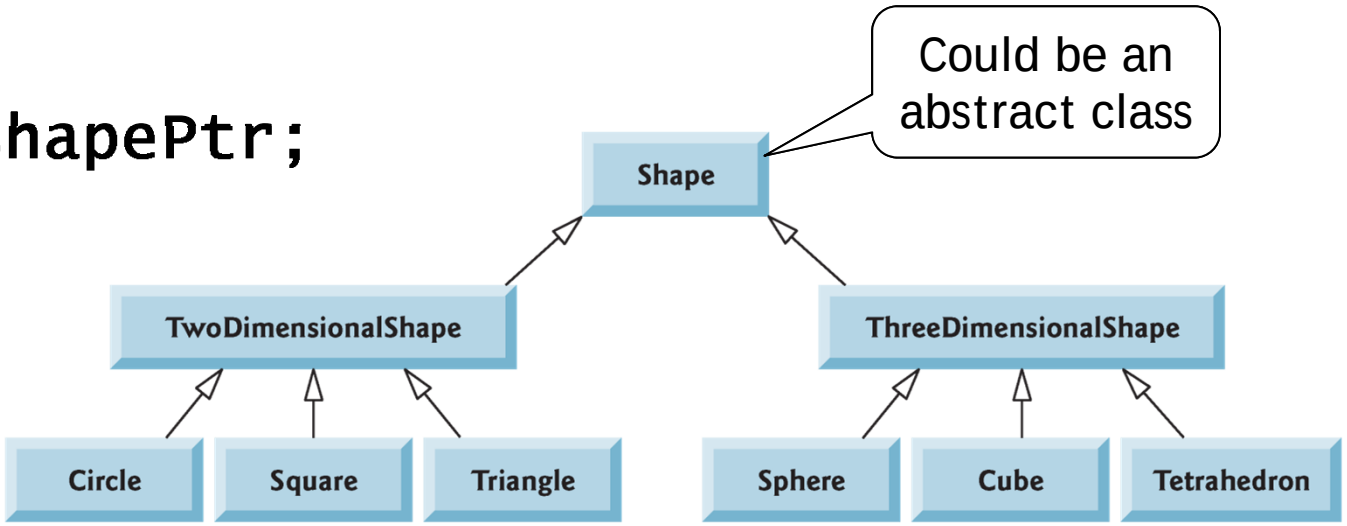
Abstract Classes (1/2)

- Abstract classes
  - Classes from which the programmer never intends to instantiate any objects
    - Incomplete—derived classes must define the “missing pieces”
    - Too generic to define real objects
  - An abstract class has at least one pure `virtual` function
- Normally used as base classes, called abstract base classes
  - Provides an appropriate base class from which other classes can inherit

Abstract Classes (2/2)

- We can use the abstract base class to declare pointers or references
  - Can refer to objects of any concrete class derived from the abstract class
  - Programs typically use such pointers and references to manipulate derived-class objects polymorphically
- Example:

```
Shape *shapePtr;
```



Concrete Classes

- Classes used to instantiate objects are called concrete classes
  - Must provide implementation for every member function they define
- Classes from which the programmer can instantiate any objects
- Every concrete derived class must override all base-class pure `virtual` functions with concrete implementations
  - If not overridden, derived-class will also be abstract (because pure virtual functions are inherited)

Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Recall: Case Study—Employees

- We'd like to design two classes for two types of employees
  - `CommissionEmployee`
    - Commission employees are paid a percentage of their sales
  - `BasePlusCommissionEmployee`
    - Base-salaried commission employees receive a base salary plus a percentage of their sales

CommissionEmployee.h

```
 9  class CommissionEmployee
10  {
11  public:
12     CommissionEmployee( const string &, const string &, const string &,
13        double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // set first name
16     string getFirstName() const; // return first name
17
18     void setLastName( const string & ); // set last name
19     string getLastName() const; // return last name
20
21     void setSocialSecurityNumber( const string & ); // set SSN
22     string getSocialSecurityNumber() const; // return SSN
23
24     void setGrossSales( double ); // set gross sales amount
25     double getGrossSales() const; // return gross sales amount
26
27     void setCommissionRate( double ); // set commission rate
28     double getCommissionRate() const; // return commission rate
29
30     double earnings() const; // calculate earnings
31     void print() const; // print CommissionEmployee object
32  private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // gross weekly sales
37     double commissionRate; // commission percentage
38  }; // end class CommissionEmployee
```

CommissionEmployee.cpp (1/2)

```
3  #include <iostream>
4  #include "CommissionEmployee.h" // CommissionEmployee class definition
5  using namespace std;
6
7  // constructor
8  CommissionEmployee::CommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate )
11    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
12 {
13    setGrossSales( sales ); // validate and store gross sales
14    setCommissionRate( rate ); // validate and store commission rate
15 } // end CommissionEmployee constructor
16
17 // set first name
18 void CommissionEmployee::setFirstName
19 {
20    firstName = first; // should valid
21 } // end function setFirstName
22
```

```
23 // return first name
24 string CommissionEmployee::getFirstName() const
25 {
26    return firstName;
27 } // end function getFirstName
28
29 // set last name
30 void CommissionEmployee::setLastName( const string &last )
31 {
32    lastName = last;  // should validate
33 } // end function setLastName
34
35 // return last name
36 string CommissionEmployee::getLastName() const
37 {
38    return lastName;
39 } // end function getLastName
40
41 // set social security number
42 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
43 {
44    socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
```

CommissionEmployee.cpp (2/2)

```
47  // return social security number
48  string CommissionEmployee::getSocialSecurityNumber() const
49  {
50     return socialSecurityNumber;
51  } // end function getSocialSecurityNumber
52
53  // set gross sales amount
54  void CommissionEmployee::setGrossSales( double sales )
55  {
56     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
57  } // end function setGrossSales
58
59  // return gross sales amount
60  double CommissionEmployee::getGrossSales() const
61  {
62     return grossSales;
63  } // end function getGrossSales
64
65  // set commission rate
66  void CommissionEmployee::setCommissionRa
67  {
68     commissionRate = ( rate > 0.0 && rate
69  } // end function setCommissionRate
70
```

```
72  double CommissionEmployee::getCommissionRate() const
73  {
74     return commissionRate;
75  } // end function getCommissionRate
76
77  // calculate earnings
78  double CommissionEmployee::earnings() const
79  {
80     return getCommissionRate() * getGrossSales();
81  } // end function earnings
82
83  // print CommissionEmployee object
84  void CommissionEmployee::print() const
85  {
86     cout << "commission employee: "
87        << getFirstName() << ' ' << getLastName()
88        << "\nsocial security number: " << getSocialSecurityNumber()
89        << "\ngross sales: " << getGrossSales()
90        << "\ncommission rate: " << getCommissionRate();
91  } // end function print
```

BasePlusCommissionEmployee.h

```
 7  #include <string> // C++ standard string class
 8  #include "CommissionEmployee.h" // CommissionEmployee class declaration
 9  using namespace std;
10
11  class BasePlusCommissionEmployee : public CommissionEmployee
12  {
13  public:
14     BasePlusCommissionEmployee( const string &, const string &,
15        const string &, double = 0.0, double = 0.0, double = 0.0 );
16
17     void setBaseSalary( double ); // set base salary
18     double getBaseSalary() const; // return base salary
19
20     double earnings() const; // calculate earnings
21     void print() const; // print BasePlusCommissionEmployee object
22  private:
23     double baseSalary; // base salary
24  }; // end class BasePlusCommissionEmployee
25
26  #endif
```

BasePlusCommissionEmployee.cpp

```
 3  #include <iostream>
 4  #include "BasePlusCommissionEmployee.h"
 5  using namespace std;
 6
 7  // constructor
 8  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
 9     const string &first, const string &last, const string &ssn,
10     double sales, double rate, double salary )
11     // explicitly call base-class constructor
12     : CommissionEmployee( first, last, ssn, sales, rate )
13  {
14     setBaseSalary( salary ); // validate and store base salary
15  } // end BasePlusCommissionEmployee constructor
16
17  // set base salary
18  void BasePlusCommissionEmployee::setBaseSalary( d
19  {
20     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21  } // end function setBaseSalary
22
```

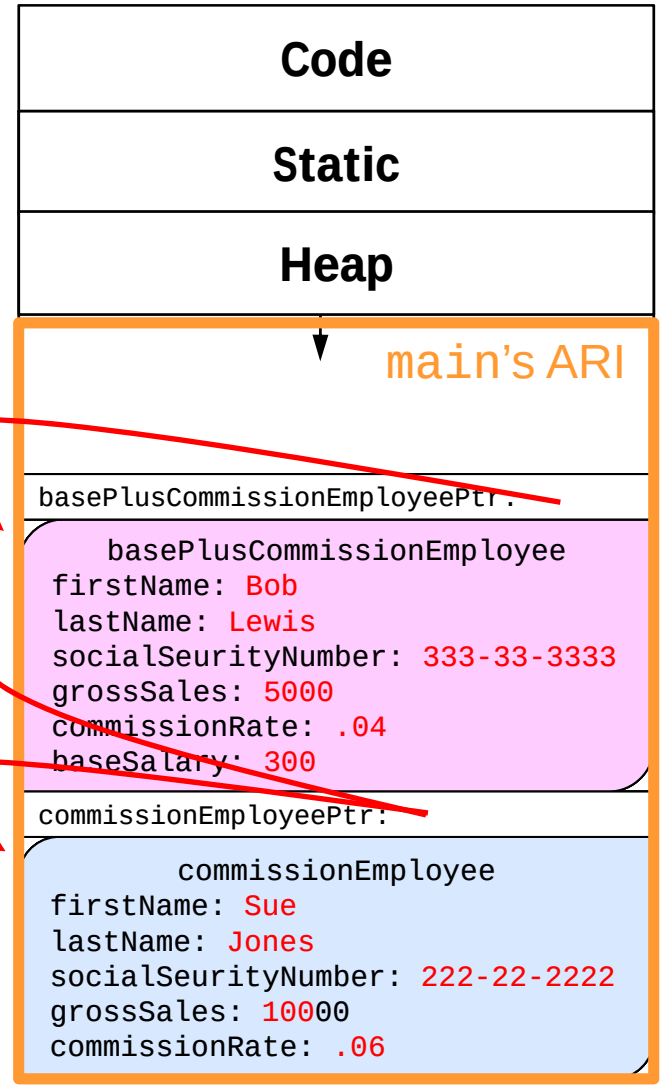
```
23  // return base salary
24  double BasePlusCommissionEmployee::getBaseSalary() const
25  {
26     return baseSalary;
27  } // end function getBaseSalary
28
29  // calculate earnings
30  double BasePlusCommissionEmployee::earnings() const
31  {
32     return getBaseSalary() + CommissionEmployee::earnings();
33  } // end function earnings
34
35  // print BasePlusCommissionEmployee object
36  void BasePlusCommissionEmployee::print() const
37  {
38     cout << "base-salaried ";
39
40     // invoke CommissionEmployee's print function
41     CommissionEmployee::print();
42
43     cout << "\nbase salary: " << getBaseSalary();
44  } // end function print
```

Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

A Client

```
4  #include <iostream>
5  #include <iomanip>
6  #include "CommissionEmployee.h"
7  #include "BasePlusCommissionEmployee.h"
8  using namespace std;
9
10 int main()
11 {
12    // create base-class object
13    CommissionEmployee commissionEmployee(
14       "Sue", "Jones", "222-22-2222", 10000, .06 );
15
16    // create base-class pointer
17    CommissionEmployee *commissionEmployeePtr = 0;
18
19    // create derived-class object
20    BasePlusCommissionEmployee basePlusCommissionEmployee(
21       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
22
23    // create derived-class pointer
24    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
25
26    // set floating-point output formatting
27    cout << fixed << setprecision( 2 );
28
29    // output objects commissionEmployee and basePlusCommissionEmployee
30    cout << "Print base-class and derived-class objects:\n\n";
31    commissionEmployee.print(); // invokes base-class print
32    cout << "\n\n";
33    basePlusCommissionEmployee.print(); // invokes derived-class print
34
35    // aim base-class pointer at base-class object and print
36    commissionEmployeePtr = &commissionEmployee; // perfectly natural
37    cout << "\n\n\nCalling print with base-class pointer to "
38       << "\nbase-class object invokes base-class print function:\n\n";
39    commissionEmployeePtr->print(); // invokes base-class print
40
41    // aim derived-class pointer at derived-class object and print
42    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
43    cout << "\n\n\nCalling print with derived-class pointer to "
44       << "\nderived-class object invokes derived-class "
45       << "print function:\n\n";
46    basePlusCommissionEmployeePtr->print(); // invokes derived-class print
47
48    // aim base-class pointer at derived-class object and print
49    commissionEmployeePtr = &basePlusCommissionEmployee;
50    cout << "\n\n\nCalling print with base-class pointer to "
51       << "derived-class object\ninvokes base-class print "
52       << "function on that derived-class object:\n\n";
53    commissionEmployeePtr->print(); // invokes base-class print
54    cout << endl;
55 } // end main
```



Results of the Previous Example

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06


Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

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

```
4  #include "CommissionEmployee.h"
5  #include "BasePlusCommissionEmployee.h"
6
7  int main()
8  {
9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13    // aim base-class pointer at derived-class object
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer (allowed)
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24    // attempt to invoke derived-class-only member functions
25    // on derived-class object through base-class pointer (disallowed)
26    double baseSalary = commissionEmployeePtr->getBaseSalary();
27    commissionEmployeePtr->setBaseSalary( 500 );
28 } // end main
```

- Allowed when the base-class pointer is explicitly casted to a derived-class pointer
  - Called downcasting

An Example of Downcasting

```
 4  #include "CommissionEmployee.h"
 5  #include "BasePlusCommissionEmployee.h"
 6
 7  int main()
 8  {
 9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10     BasePlusCommissionEmployee basePlusCommissionEmployee(
11        "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13     // aim base-class pointer at derived-class object
14     commissionEmployeePtr = &basePlusCommissionEmployee;
15
16     // invoke base-class member functions on derived-class
17     // object through base-class pointer (allowed)
18     string firstName = commissionEmployeePtr->getFirstName();
19     string lastName = commissionEmployeePtr->getLastName();
20     string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21     double grossSales = commissionEmployeePtr->getGrossSales();
22     double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24     BasePlusCommissionEmployee *derivedPtr =
25        dynamic_cast<BasePlusCommissionEmployee *>(commissionEmployeePtr);
26     if ( derivedPtr != 0) { // 0 if not a BasePlusCommissionEmployee
27       derivedPtr->setBaseSalary( 500 );
28     }
   } // end main
```

dynamic_cast vs static_cast

- You cannot convert a base-class pointer/reference to a derived-class pointer/reference unless
  - the base class has a conversion operator, or
  - the derived class has a conversion constructor

```
int main() {
  derivedClass d;

  baseClass &bRef = d;
  derivedClass &dRef1 = static_cast<derivedClass &>(bRef);
  derivedClass &dRef2 = dynamic_cast<derivedClass &>(bRef);

  return 0;
}
```

`static_cast` is more dangerous; there is no check that the conversion is valid, so if `bRef` doesn't refer to a `derivedClass` object, using `dRef1` will have undefined behavior

`dynamic_cast` is safer, but will only work if `baseClass` is polymorphic (that is, if it has a virtual function). If `bRef` refers to an object of incompatible type, it will throw an exception

Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Aiming Derived-Class Pointers at Base-Class Objects

```
1  // Fig. 13.6: fig13_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main()
7  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15 } // end main
```

```
basePlusCommissionEmployeePtr =
   dynamic_cast<BasePlusCommissionEmployee *>(&commissionEmployee);
```

Telling the compiler that I know that what I'm doing is dangerous and I take full responsibility for my actions

Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

CommissionEmployee.h

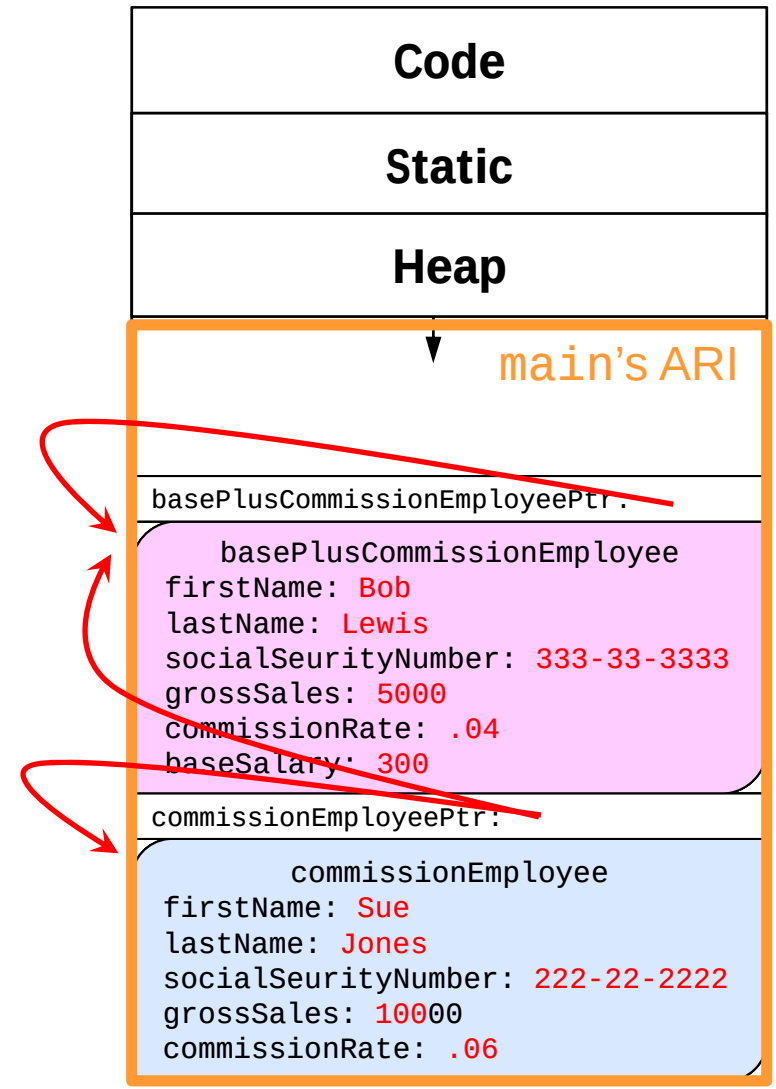
```
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    virtual double earnings() const; // calculate earnings
31    virtual void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
```

BasePlusCommissionEmployee.h

```
 7  #include <string> // C++ standard string class
 8  #include "CommissionEmployee.h" // CommissionEmployee class declaration
 9  using namespace std;
10
11  class BasePlusCommissionEmployee : public CommissionEmployee
12  {
13  public:
14     BasePlusCommissionEmployee( const string &, const string &,
15        const string &, double = 0.0, double = 0.0, double = 0.0 );
16
17     void setBaseSalary( double ); // set base salary
18     double getBaseSalary() const; // return base salary
19
20     virtual double earnings() const; // calculate earnings
21     virtual void print() const; // print BasePlusCommissionEmployee object
22  private:
23     double baseSalary; // base salary
24  }; // end class BasePlusCommissionEmployee
```

A Client

```
3  #include <iostream>
4  #include <iomanip>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main()
10 {
11    // create base-class object
12    CommissionEmployee commissionEmployee(
13       "Sue", "Jones", "222-22-2222", 10000, .06 );
14
15    // create base-class pointer
16    CommissionEmployee *commissionEmployeePtr = 0;
17
18    // create derived-class object
19    BasePlusCommissionEmployee basePlusCommissionEmployee(
20       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
21
22    // create derived-class pointer
23    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
24
25    // set floating-point output formatting
26    cout << fixed << setprecision( 2 );
27
28    // output objects using static binding
29    cout << "Invoking print function on base-class and derived-class "
30       << "\nobjects with static binding\n\n";
31    commissionEmployee.print(); // static binding
32    cout << "\n\n";
33    basePlusCommissionEmployee.print(); // static binding
34
35    // output objects using dynamic binding
36    cout << "\n\n\nInvoking print function on base-class and "
37       << "derived-class \nobjects with dynamic binding";
38
39    // aim base-class pointer at base-class object and print
40    commissionEmployeePtr = &commissionEmployee;
41    cout << "\n\nCalling virtual function print with base-class pointer"
42       << "\nto base-class object invokes base-class "
43       << "print function:\n\n";
44    commissionEmployeePtr->print(); // invokes base-class print
45
46    // aim derived-class pointer at derived-class object and print
47    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
48    cout << "\n\nCalling virtual function print with derived-class "
49       << "pointer\nto derived-class object invokes derived-class "
50       << "print function:\n\n";
51    basePlusCommissionEmployeePtr->print(); // invokes derived-class print
52
53    // aim base-class pointer at derived-class object and print
54    commissionEmployeePtr = &basePlusCommissionEmployee;
55    cout << "\n\nCalling virtual function print with base-class pointer"
56       << "\nto derived-class object invokes derived-class "
57       << "print function:\n\n";
58
59    // polymorphism; invokes BasePlusCommissionEmployee's print;
60    // base-class pointer to derived-class object
61    commissionEmployeePtr->print();
62    cout << endl;
63 } // end main
```



Results of the Previous Example

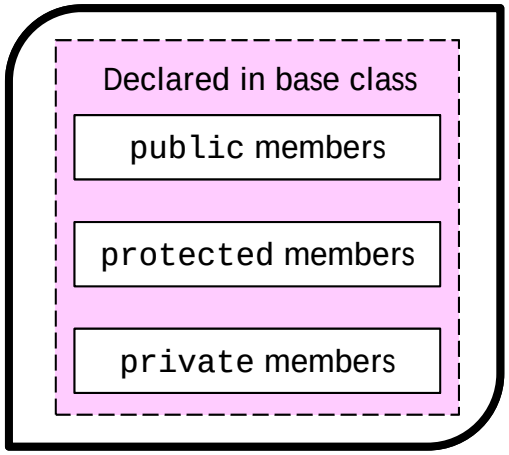
```
Invoking print function on base-class and derived-class
objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Invoking print function on base-class and derived-class
objects with dynamic binding

Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Outline

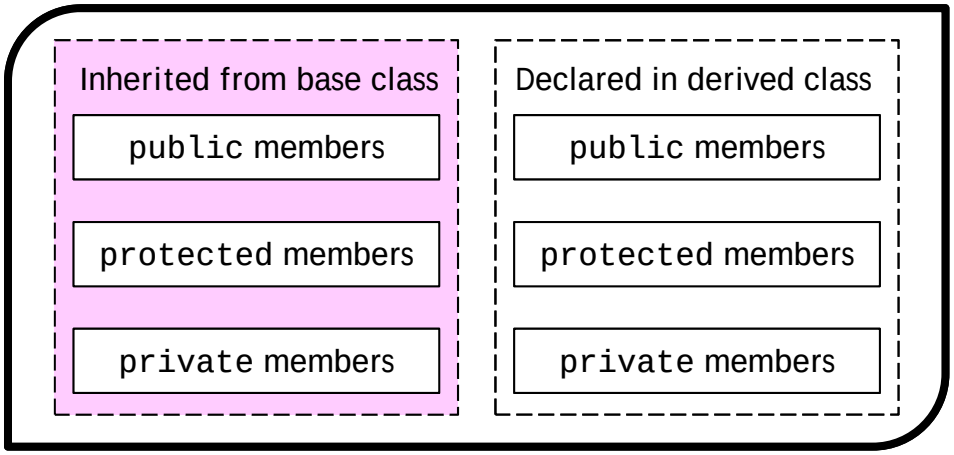
- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Base-Class Objects vs Derived-Class Objects

- Derived-class objects can be treated as if they were base-class objects (when using `public` inheritance)
- Base-class objects cannot be treated as if they were derived-class objects



Base-class object



Derived-class objects
(can be treated as a base-class object)

Pointers and Objects between Classes

Aiming	At	
	Base-class object	**Derived-class object**
Base-class pointer	Calling base-class redefined functions	Calling base-class redefined functions
	Calling base-class `virtual` functions	Calling derived-class `virtual` functions
Derived-class pointer	ERROR (without explicit cast)	Calling derived-class redefined functions
		Calling derived-class `virtual` functions

Polymorphism

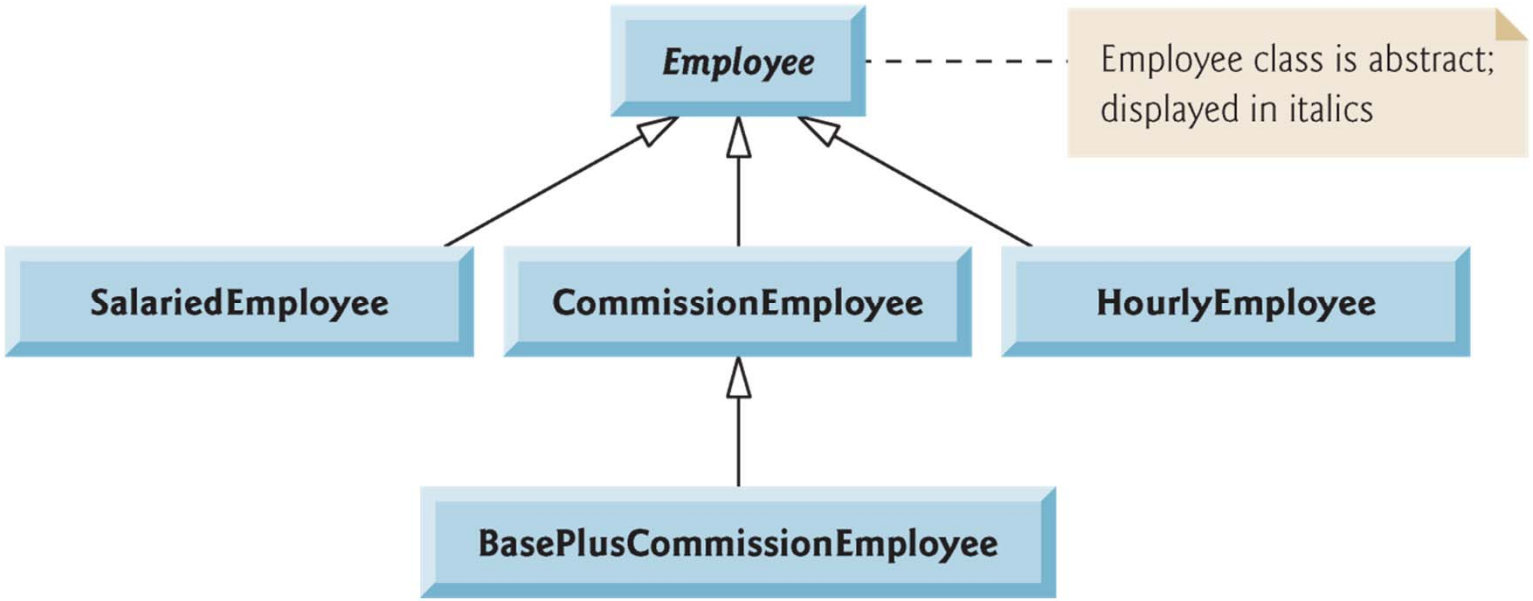
Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Case Study: Writing a Payroll System

- Employees are paid weekly
- Four types of employees
  - Salaried employee
    - Paid a fixed weekly salary
  - Hourly employees
    - Paid by the hour and receive overtime (exceeding 40 hours) pay
  - Commission employees
    - Paid a percentage of their sales
  - Base-salary-plus-commission employees
    - Receive a base salary plus a percentage of their sales

Designing the Class Hierarchy



Employee.h

```
 6  #include <string> // C++ standard string class
 7  using namespace std;
 8
 9  class Employee
10  {
11  public:
12     Employee( const string &, const string &, const string & );
13
14     void setFirstName( const string & ); // set first name
15     string getFirstName() const; // return first name
16
17     void setLastName( const string & ); // set last name
18     string getLastName() const; // return last name
19
20     void setSocialSecurityNumber( const string & ); // set SSN
21     string getSocialSecurityNumber() const; // return SSN
22
23     // pure virtual function makes Employee abstract base class
24     virtual double earnings() const = 0; // pure virtual
25     virtual void print() const; // virtual
26  private:
27     string firstName;
28     string lastName;
29     string socialSecurityNumber;
30  }; // end class Employee
```

Polymorphic Interface

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Employee.cpp

```
4  #include <iostream>
5  #include "Employee.h" // Employee class definition
6  using namespace std;
7
8  // constructor
9  Employee::Employee( const string &first, const string &last,
10    const string &ssn )
11    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
12 {
13    // empty body
14 } // end Employee constructor
15
16 // set first name
17 void Employee::setFirstName( const string &first )
18 {
19    firstName = first;
20 } // end function setFirstName
21
22 // return first name
23 string Employee::getFirstName() const
24 {
25    return firstName;
26 } // end function getFirstName
27
28 // set last name
29 void Employee::setLastName( const string &last )
30 {
31    lastName = last;
32 } // end function setLastName
33
34 // return last name
35 string Employee::getLastName() const
36 {
37    return lastName;
38 } // end function getLastName
39
40 // set social security number
41 void Employee::setSocialSecurityNumber( const string &ssn )
42 {
43    socialSecurityNumber = ssn; // should validate
44 } // end function setSocialSecurityNumber
45
46 // return social security number
47 string Employee::getSocialSecurityNumber() const
48 {
49    return socialSecurityNumber;
50 } // end function getSocialSecurityNumber
51
52 // print Employee's information (virtual, but not pure virtual)
53 void Employee::print() const
54 {
55    cout << getFirstName() << ' ' << getLastName()
56       << "\nsocial security number: " << getSocialSecurityNumber();
57 } // end function print
```

SalariedEmployee.h

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

Good programming practice

SalariedEmployee.cpp

```
3  #include <iostream>
4  #include "SalariedEmployee.h" // SalariedEmployee class definition
5  using namespace std;
6
7  // constructor
8  SalariedEmployee::SalariedEmployee( const string &first,
9     const string &last, const string &ssn, double salary )
10    : Employee( first, last, ssn )
11 {
12    setWeeklySalary( salary );
13 } // end SalariedEmployee constructor
14
15 // set salary
16 void SalariedEmployee::setWeeklySalary( double salary )
17 {
18    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
19 } // end function setWeeklySalary
20
21 // return salary
22 double SalariedEmployee::getWeeklySalary() const
23 {
24    return weeklySalary;
25 } // end function getWeeklySalary
26
27 // calculate earnings;
28 // override pure virtual function earnings in Employee
29 double SalariedEmployee::earnings() const
30 {
31    return getWeeklySalary();
32 } // end function earnings
33
34 // print SalariedEmployee's information
35 void SalariedEmployee::print() const
36 {
37    cout << "salaried employee: ";
38    Employee::print(); // reuse abstract base-class print function
39    cout << "\nweekly salary: " << getWeeklySalary();
40 } // end function print
```

HourlyEmployee.h

```
 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     static const int hoursPerWeek = 168; // hours in one week
12
13     HourlyEmployee( const string &, const string &,
14        const string &, double = 0.0, double = 0.0 );
15
16     void setWage( double ); // set hourly wage
17     double getWage() const; // return hourly wage
18
19     void setHours( double ); // set hours worked
20     double getHours() const; // return hours worked
21
22     // keyword virtual signals intent to override
23     virtual double earnings() const; // calculate earnings
24     virtual void print() const; // print HourlyEmployee object
25  private:
26     double wage; // wage per hour
27     double hours; // hours worked for week
28  }; // end class HourlyEmployee
29
30  #endif // HOURLY_H
```

HourlyEmployee.cpp

```
3  #include <iostream>
4  #include "HourlyEmployee.h" // HourlyEmployee class definition
5  using namespace std;
6
7  // constructor
8  HourlyEmployee::HourlyEmployee( const string &first, const string &last,
9     const string &ssn, double hourlyWage, double hoursWorked )
10    : Employee( first, last, ssn )
11 {
12    setWage( hourlyWage ); // validate hourly wage
13    setHours( hoursWorked ); // validate hours worke
14 } // end HourlyEmployee constructor
15
16 // set wage
17 void HourlyEmployee::setWage( double hourlyWage )
18 {
19    wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
20 } // end function setWage
21
22 // return wage
23 double HourlyEmployee::getWage() const
24 {
25    return wage;
26 } // end function getWage
27
28 // set hours worked
29 void HourlyEmployee::setHours( double hoursWorked )
30 {
31    hours = ( ( ( hoursWorked >= 0.0 ) &&
32       ( hoursWorked <= hoursPerWeek ) ) ? hoursWorked : 0.0 );
33 } // end function setHours
34
35 // return hours worked
36 double HourlyEmployee::getHours() const
37 {
38    return hours;
39 } // end function getHours
40
```

```
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double HourlyEmployee::earnings() const
44 {
45    if ( getHours() <= 40 ) // no overtime
46       return getWage() * getHours();
47    else
48       return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
49 } // end function earnings
50
51 // print HourlyEmployee's information
52 void HourlyEmployee::print() const
53 {
54    cout << "hourly employee: ";
55    Employee::print(); // code reuse
56    cout << "\nhourly wage: " << getWage() <<
57       "; hours worked: " << getHours();
58 } // end function print
```

CommissionEmployee.h

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print CommissionEmployee object
23 private:
24    double grossSales; // gross weekly sales
25    double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

CommissionEmployee.cpp

```
3  #include <iostream>
4  #include "CommissionEmployee.h" // CommissionEmployee class definition
5  using namespace std;
6
7  // constructor
8  CommissionEmployee::CommissionEmployee( const string &first,
9     const string &last, const string &ssn, double sales, double rate )
10    : Employee( first, last, ssn )
11 {
12    setGrossSales( sales );
13    setCommissionRate( rate );
14 } // end CommissionEmployee constructor
15
16 // set commission rate
17 void CommissionEmployee::setCommissionRate( double rate )
18 {
19    commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
20 } // end function setCommissionRate
21
22 // return commission rate
23 double CommissionEmployee::getCommissionRate() const
24 {
25     return commissionRate;
26 } // end function getCommissionRate
27
28 // set gross sales amount
29 void CommissionEmployee::setGrossSales( double sales )
30 {
31    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
32 } // end function setGrossSales
33
34 // return gross sales amount
35 double CommissionEmployee::getGrossSales() const
36 {
37     return grossSales;
38 } // end function getGrossSales
39
40 // calculate earnings; override pure virtual function earnings in Employee
41 double CommissionEmployee::earnings() const
42 {
43    return getCommissionRate() * getGrossSales();
44 } // end function earnings
45
46 // print CommissionEmployee's information
47 void CommissionEmployee::print() const
48 {
49    cout << "commission employee: ";
50    Employee::print(); // code reuse
51    cout << "\ngross sales: " << getGrossSales()
52       << "; commission rate: " << getCommissionRate();
53 } // end function print
```

BasePlusCommissionEmployee.h

```
1  // Fig. 13.21: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from CommissionEmployee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  class BasePlusCommissionEmployee : public CommissionEmployee
9  {
10 public:
11    BasePlusCommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14    void setBaseSalary( double ); // set base salary
15    double getBaseSalary() const; // return base salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print BasePlusCommissionEmployee object
20 private:
21    double baseSalary; // base salary per week
22 }; // end class BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H
```

BasePlusCommissionEmployee.cpp

```
3  #include <iostream>
4  #include "BasePlusCommissionEmployee.h"
5  using namespace std;
6
7  // constructor
8  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate, double salary )
11    : CommissionEmployee( first, last, ssn, sales, rate )
12 {
13    setBaseSalary( salary ); // validate and store base salary
14 } // end BasePlusCommissionEmployee constructor
15
16 // set base salary
17 void BasePlusCommissionEmployee::setBaseSalary( double salary )
18 {
19    baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
20 } // end function setBaseSalary
21
22 // return base salary
23 double BasePlusCommissionEmployee::getBaseSalary() const
24 {
25     return baseSalary;
26 } // end function getBaseSalary
27
28 // calculate earnings;
29 // override virtual function earnings in CommissionEmployee
30 double BasePlusCommissionEmployee::earnings() const
31 {
32     return getBaseSalary() + CommissionEmployee::earnings();
33 } // end function earnings
34
35 // print BasePlusCommissionEmployee's information
36 void BasePlusCommissionEmployee::print() const
37 {
38    cout << "base-salaried ";
39    CommissionEmployee::print(); // code reuse
40    cout << "; base salary: " << getBaseSalary();
41 } // end function print
```

Demonstrating Polymorphic Processing (1/2)

```
4  #include <iostream>
5  #include <iomanip>
6  #include <vector>
7  #include "Employee.h"
8  #include "SalariedEmployee.h"
9  #include "HourlyEmployee.h"
10 #include "CommissionEmployee.h"
11 #include "BasePlusCommissionEmployee.h"
12 using namespace std;
13
14 void virtualViaPointer( const Employee * const ); // prototype
15 void virtualViaReference( const Employee & ); // prototype
16
17 int main()
18 {
19    // set floating-point output formatting
20    cout << fixed << setprecision( 2 );
21
22    // create derived-class objects
23    SalariedEmployee salariedEmployee(
24       "John", "Smith", "111-11-1111", 800 );
25    HourlyEmployee hourlyEmployee(
26       "Karen", "Price", "222-22-2222", 16.75, 40 );
27    CommissionEmployee commissionEmployee(
28       "Sue", "Jones", "333-33-3333", 10000, .06 );
29    BasePlusCommissionEmployee basePlusCommissionEmployee(
30       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
31
32    cout << "Employees processed individually using static binding:\n\n";
33
34    // output each Employee's information and earnings using static binding
35    salariedEmployee.print();
36    cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
37    hourlyEmployee.print();
38    cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
39    commissionEmployee.print();
40    cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
41    basePlusCommissionEmployee.print();
42    cout << "\nearned $" << basePlusCommissionEmployee.earnings()
43       << "\n\n";
44
```

Demonstrating Polymorphic Processing (2/2)

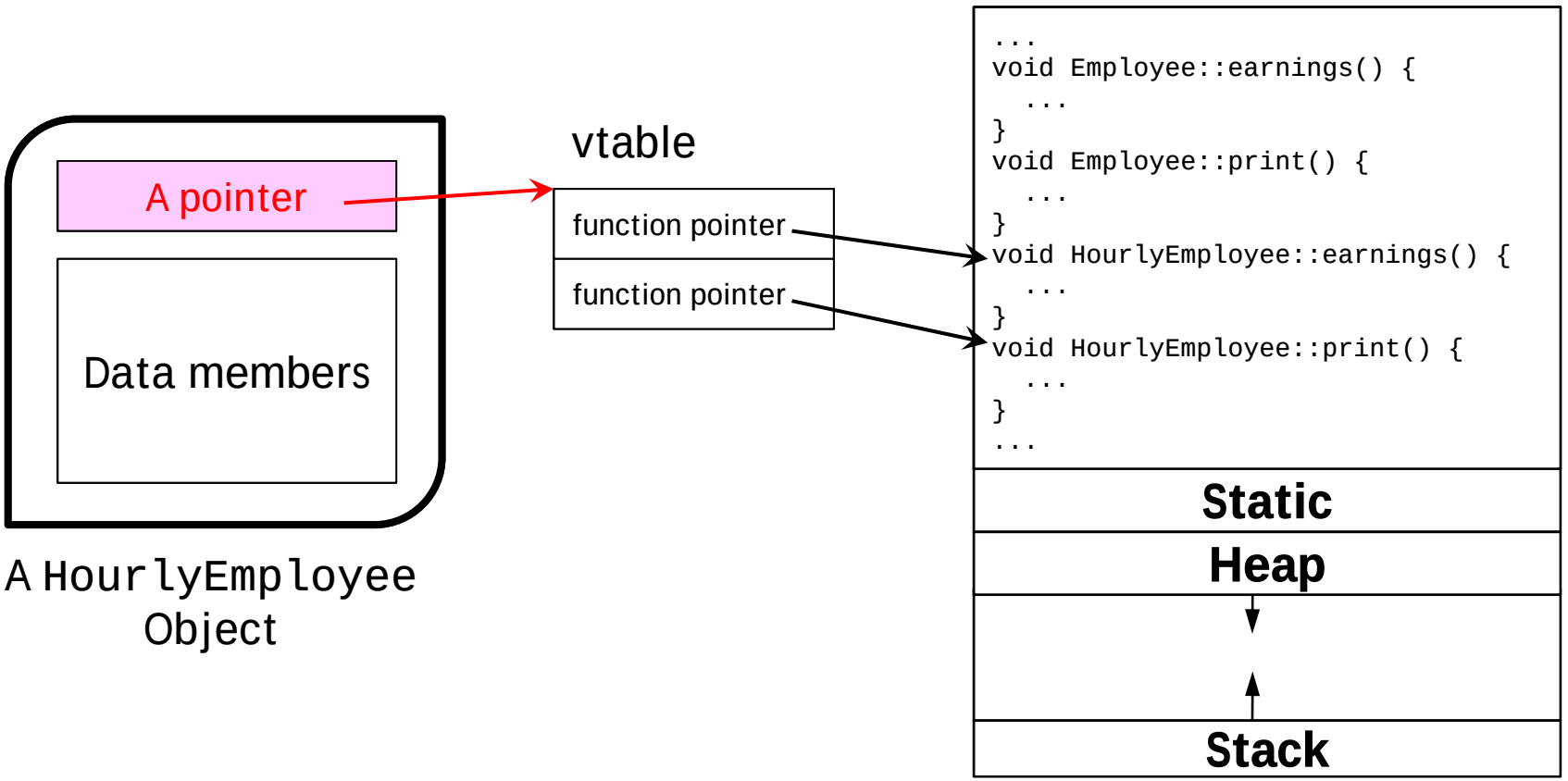
```
45      // create vector of four base-class pointers
46      vector < Employee * > employees( 4 );
47
48      // initialize vector with Employees
49      employees[ 0 ] = &salariedEmployee;
50      employees[ 1 ] = &hourlyEmployee;
51      employees[ 2 ] = &commissionEmployee;
52      employees[ 3 ] = &basePlusCommissionEmployee;
53
54      cout << "Employees processed polymorphically via dynamic binding:\n\n";
55
56      // call virtualViaPointer to print each Employee's information
57      // and earnings using dynamic binding
58      cout << "Virtual function calls made off base-class pointers:\n\n";
59
60      for ( size_t i = 0; i < employees.size(); i++ )
61         virtualViaPointer( employees[ i ] );
62
63      // call virtualViaReference to print each Employee's information
64      // and earnings using dynamic binding
65      cout << "Virtual function calls made off base-class references:\n\n";
66
67      for ( size_t i = 0; i < employees.size(); i++ )
68         virtualViaReference( *employees[ i ] ); // note dereferencing
69   } // end main
70
71   // call Employee virtual functions print and earnings off a
72   // base-class pointer using dynamic binding
73   void virtualViaPointer( const Employee * const baseClassPtr )
74   {
75      baseClassPtr->print();
76      cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
77   } // end function virtualViaPointer
78
79   // call Employee virtual functions print and earnings off a
80   // base-class reference using dynamic binding
81   void virtualViaReference( const Employee &baseClassRef )
82   {
83      baseClassRef.print();
84      cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
85   } // end function virtualViaReference
```

Outline

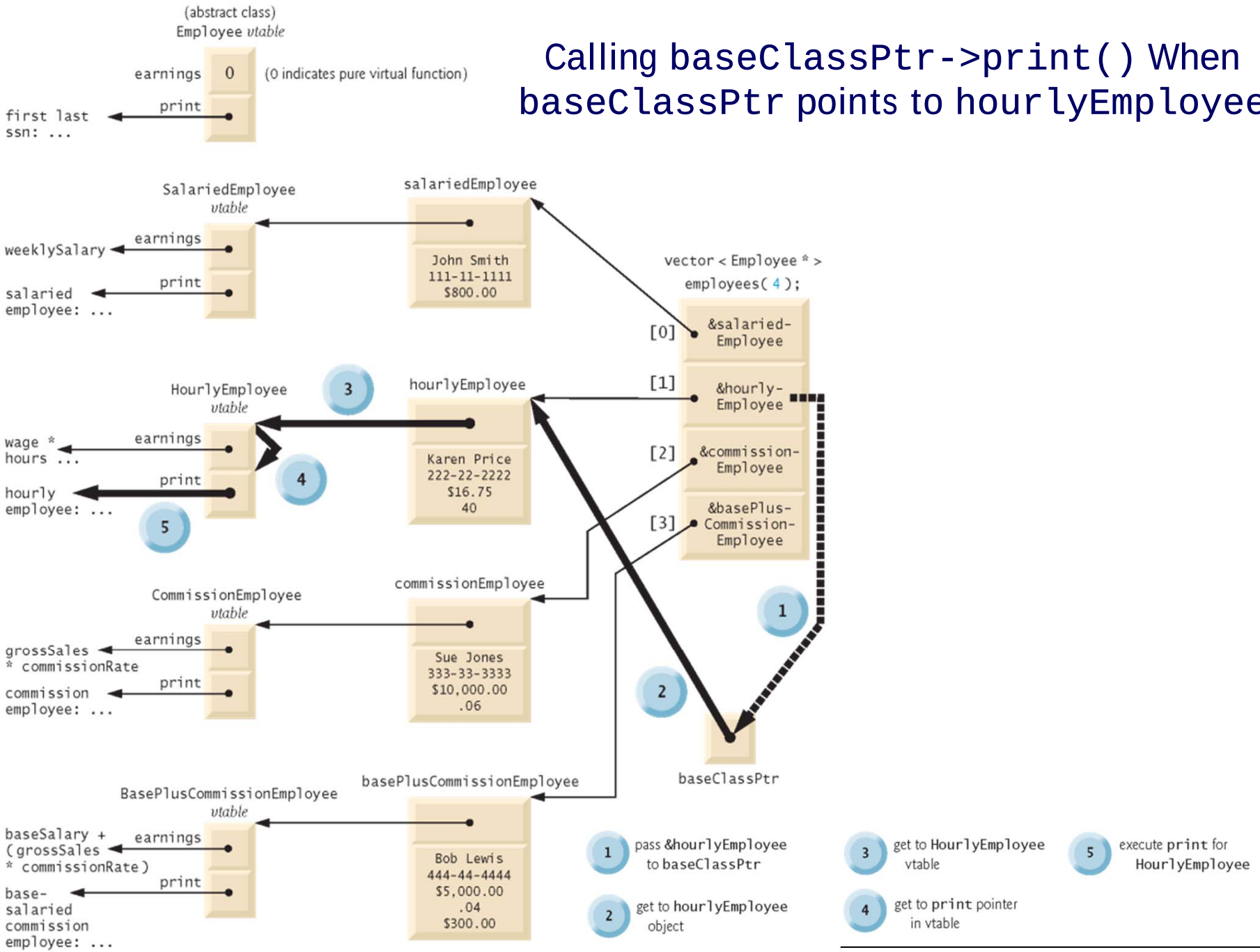
- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Objects Calling `virtual` Functions

- Each object of a class that has one or more `virtual` functions *has a pointer* to its virtual function table (*vtable*)



Calling `baseClassPtr->print()` When `baseClassPtr` points to `hourlyEmployee`



Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Runtime Type Information (RTTI)

- Operator `typeid` returns a reference to an object of class `type_info` that contains the information about the type of its operand
- Member function `name` (of class `type_info`) returns a pointer-based string that contains the type name
- To use `typeid`, the program must include header file `<typeinfo>`
- Some compilers require that RTTI be enabled before it can be used in a program
  - In Visual C++ 2008, this option is enabled by default

Downcasting and Runtime Type Information (1/3)

```
 5  #include <iostream>
 6  #include <iomanip>
 7  #include <vector>
 8  #include <typeinfo>
 9  #include "Employee.h"
10  #include "SalariedEmployee.h"
11  #include "HourlyEmployee.h"
12  #include "CommissionEmployee.h"
13  #include "BasePlusCommissionEmployee.h"
14  using namespace std;
15
16  int main()
17  {
18     // set floating-point output formatting
19     cout << fixed << setprecision( 2 );
20
21     // create vector of four base-class pointers
22     vector < Employee * > employees( 4 );
23
24     // initialize vector with various kinds of Employees
25     employees[ 0 ] = new SalariedEmployee(
26        "John", "Smith", "111-11-1111", 800 );
27     employees[ 1 ] = new HourlyEmployee(
28        "Karen", "Price", "222-22-2222", 16.75, 40 );
29     employees[ 2 ] = new CommissionEmployee(
30        "Sue", "Jones", "333-33-3333", 10000, .06 );
31     employees[ 3 ] = new BasePlusCommissionEmployee(
32        "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
33
```

Downcasting and Runtime Type Information (2/3)

```
34      // polymorphically process each element in vector employees
35      for ( size_t i = 0; i < employees.size(); i++ )
36      {
37         employees[ i ]->print(); // output employee information
38         cout << endl;
39
40         // downcast pointer
41         BasePlusCommissionEmployee *derivedPtr =
42            dynamic_cast < BasePlusCommissionEmployee * >
43               ( employees[ i ] );
44
45         // determine whether element points to base-salaried
46         // commission employee
47         if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
48         {
49            double oldBaseSalary = derivedPtr->getBaseSalary();
50            cout << "old base salary: $" << oldBaseSalary << endl;
51            derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
52            cout << "new base salary with 10% increase is: $"
53               << derivedPtr->getBaseSalary() << endl;
54         } // end if
55
56         cout << "earned $" << employees[ i ]->earnings() << "\n\n";
57      } // end for
58
59      // release objects pointed to by vector's elements
60      for ( size_t j = 0; j < employees.size(); j++ )
61      {
62         // output class name
63         cout << "deleting object of "
64            << typeid( *employees[ j ] ).name() << endl;
65
66         delete employees[ j ];
67      } // end for
68   } // end main
```

Downcasting and Runtime Type Information (3/3)

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

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

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 HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

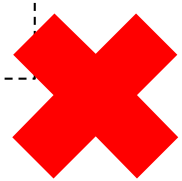
Outline

- Concept of Polymorphism
  - Virtual Functions
    - Pure Virtual Functions
    - Abstract Classes
    - Concrete Classes
- Case Study: Employees
  - Aiming Base-Class Pointers at Derived-Class Objects
    - Downcasting
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Using Virtual Functions
- Summary of Bases and Derived Classes
- Case Study: Payroll Systems
  - How `virtual` Function Calls Work
- Runtime Type Information
- `virtual` Destructors

Nonvirtual Destructors

- Nonvirtual destructors
  - Destructors that are not declared with keyword `virtual`
  - If a derived-class object is destroyed explicitly by applying the **`delete`** operator to a base-class pointer to the object, the behavior is undefined
- For example

```
Shape * bPtr = new Circle();
delete bPtr;
```



`virtual` Destructors

- Declared with keyword `virtual` in the base class
  - All derived-class destructors are `virtual`
    - Even though they do not have the same name as the base-class constructor
- If a derived-class object is destroyed explicitly by applying the `delete` operator to a base-class pointer to the object, the appropriate derived-class destructor is called
  - Appropriate base-class destructor(s) will execute afterwards

`virtual` Destructors: Tip

- If a class has `virtual` functions, provide a `virtual` destructor, even if one is not required for the class
  - A custom derived-class destructor (if there is one) will be invoked when a derived-class object is deleted via s base-class pointer
- Constructors cannot be `virtual`
  - Declaring a constructor `virtual` is a compilation error