```
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
```

Notice that the base salary is *not* displayed

Fig. 12.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 5 of 5.)

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

Aiming a Base-Class Pointer at a Base-Class Object

- Line 36 assigns the address of base-class object `commissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 39 uses to invoke member function `print` on that `CommissionEmployee` object.
  - This invokes the version of `print` defined in base class `CommissionEmployee`.

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

Aiming a Derived-Class Pointer at a Derived-Class Object

- Line 42 assigns the address of derived-class object `basePlusCommissionEmployee` to derived-class pointer `basePlusCommissionEmployee-Ptr`, which line 46 uses to invoke member function `print` on that `BasePlusCommissionEmployee` object.
  - This invokes the version of `print` defined in derived class `BasePlusCommissionEmployee`.

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

Aiming a Base-Class Pointer at a Derived-Class Object

- Line 49 assigns the address of derived-class object `BasePlusCommissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 53 uses to invoke member function `print`.
  - This "crossover" is allowed because an object of a derived class *is an* object of its base class.
  - Note that despite the fact that the base class `CommissionEmployee` pointer points to a derived class `BasePlusCommissionEmployee` object, the base class `CommissionEmployee`'s `print` member function is invoked (rather than `BasePlusCommissionEmployee`'s `print` function).
- The output of each `print` member-function invocation in this program reveals that *the invoked functionality depends on the type of the pointer (or reference) used to invoke the function, not the type of the object for which the member function is called.*

12.3.2 Aiming Derived-Class Pointers at Base-Class Objects

- In Fig. 12.2, we aim a derived-class pointer at a base-class object.
- Line 14 attempts to assign the address of base-class object `commissionEmployee` to derived-class pointer `basePlusCommissionEmployeePtr`, but the C++ compiler generates an error.
- The compiler prevents this assignment, because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`.

```
1  // Fig. 12.2: fig12_02.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 = nullptr;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15 } // end main
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp8_examples\ch12\Fig12_02\fig12_02.cpp(14): error C2440: '=' :
   cannot convert from 'CommissionEmployee *' to 'BasePlusCommissionEmployee *'
          Cast from base to derived requires dynamic_cast or static_cast
```

Fig. 12.2 | Aiming a derived-class pointer at a base-class object.

12.3.3 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.
- Figure 12.3 shows the consequences of attempting to invoke a derived-class member function off a base-class pointer.

```
1  // Fig. 12.3: fig12_03.cpp
2  // Attempting to invoke derived-class-only member functions
3  // via a base-class pointer.
4  #include <string>
5  #include "CommissionEmployee.h"
6  #include "BasePlusCommissionEmployee.h"
7  using namespace std;
8
9  int main()
10 {
11    CommissionEmployee *commissionEmployeePtr = nullptr; // base class ptr
12    BasePlusCommissionEmployee basePlusCommissionEmployee(
13       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
14
15    // aim base-class pointer at derived-class object (allowed)
16    commissionEmployeePtr = &basePlusCommissionEmployee;
17
```

Fig. 12.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 1 of 2.)

```
18      // invoke base-class member functions on derived-class
19      // object through base-class pointer (allowed)
20      string firstName = commissionEmployeePtr->getFirstName();
21      string lastName = commissionEmployeePtr->getLastName();
22      string ssn = commissionEmployeePtr->getSocialSecurityNumber();
23      double grossSales = commissionEmployeePtr->getGrossSales();
24      double commissionRate = commissionEmployeePtr->getCommissionRate();
25
26      // attempt to invoke derived-class-only member functions
27      // on derived-class object through base-class pointer (disallowed)
28      double baseSalary = commissionEmployeePtr->getBaseSalary();
29      commissionEmployeePtr->setBaseSalary( 500 );
30  } // end main
```

GNU C++ compiler error messages:

```
fig12_03.cpp:28:47: error: 'class CommissionEmployee' has no member named
   'getBaseSalary'
fig12_03.cpp:29:27: error: 'class CommissionEmployee' has no member named
   'setBaseSalary'
```

Fig. 12.3 | Attempting to invoke derived-class-only functions via a base-class pointer. (Part 2 of 2.)

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

Downcasting

- 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.

Software Engineering Observation 12.3

If the address of a derived-class object has been assigned to a pointer of one of its direct or indirect base classes, it's acceptable to cast that base-class pointer back to a pointer of the derived-class type. In fact, this must be done to call derived-class member functions that do not appear in the base class.

12.3.4 Virtual Functions and Virtual Destructors

Why virtual Functions Are Useful

- Consider why `virtual` functions are useful: Suppose that 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`, but 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 generally as objects of the base class `Shape`.
  - To draw any shape, we could simply use a base-class `Shape` pointer to invoke function `draw` and 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.
  - This is *polymorphic behavior*.

Software Engineering Observation 12.4

With `virtual` functions, the type of the object, not the type of the handle used to invoke the member function, determines which version of a `virtual` function to invoke.

12.3.4 Virtual Functions and Virtual Destructors (cont.)

Declaring virtual Functions

- To enable this behavior, we declare `draw` in the base class as a `virtual` function, and we 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.

Software Engineering Observation 12.5

Once a function is declared `virtual`, it remains `virtual` all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared `virtual` when a derived class overrides it.