

Abstraction: simplify the description of something to those aspects that are relevant to the problem at hand.

Generalization: find and exploit the common properties in a set of abstractions.

hierarchy

polymorphism

genericity

patterns

Hierarchy:

Exploitation of an “is-a-kind-of” relationship among kinds of entities to allow related kinds to share properties and implementation.

Polymorphism:

Exploitation of logical or structural similarities of organization to allow related kinds to exhibit similar behaviors via similar interfaces.

Genericity:

Exploitation of logical or structural similarities of organization to produce generic objects.

Patterns:

Exploitation of common relationship scenarios among objects. (e.g., client/server system)

Represented by generalize/specialize graph

Based on “is-a-kind-of” relationship

E.g., a Manager is an Employee; a robin is a bird, and so is an ostrich.

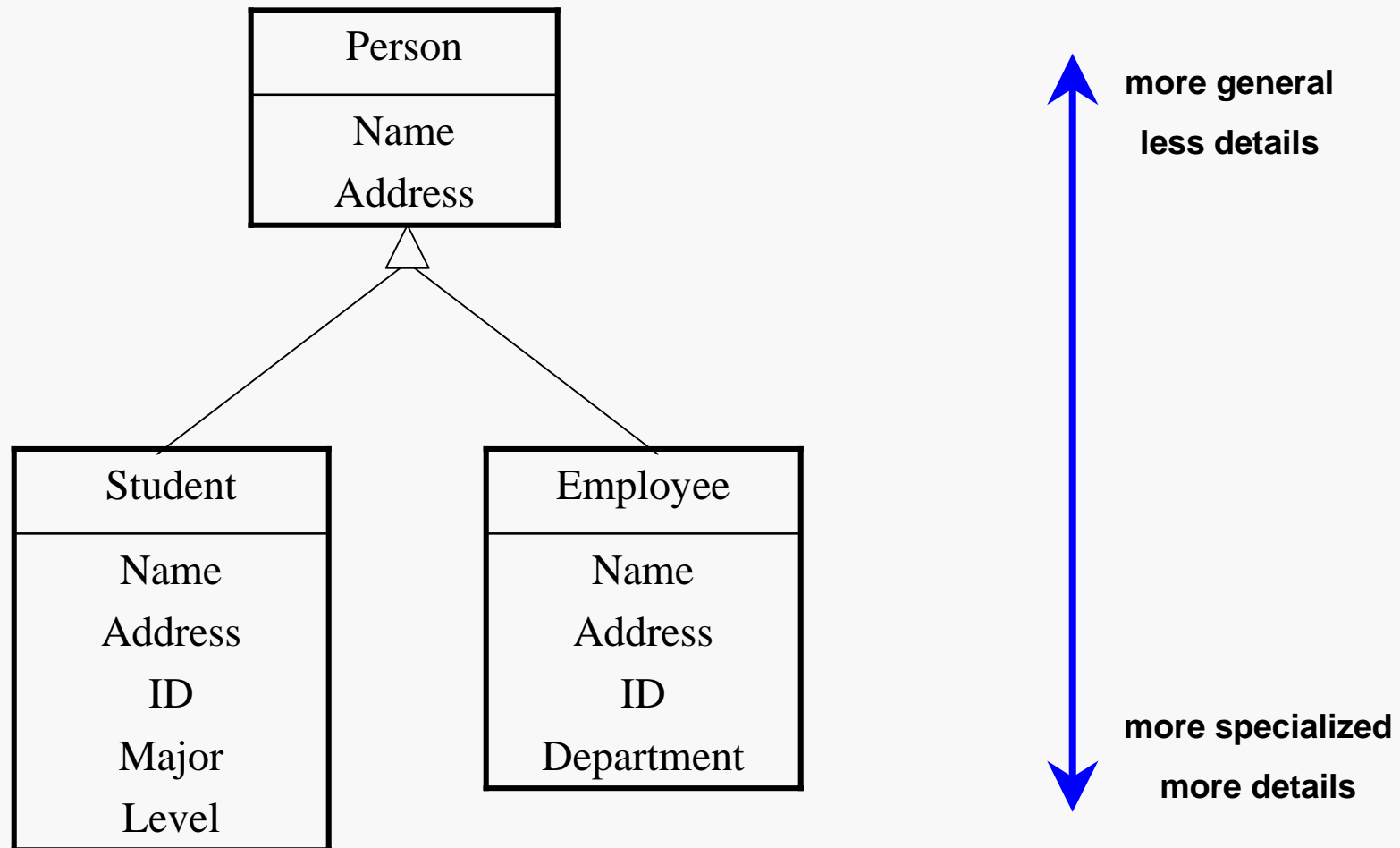
Is a form of knowledge representation – a “taxonomy” structures knowledge about nearby entities.

Extendable without redefining everything

E.g., knowing a robin is a bird tells me that a robin has certain properties and behaviors, assuming I know what a “bird” is.

Specialization can be added to proper subset of hierarchy

A generalization/specialization hierarchy based on “is-a-kind-of” relationships:



Terminology

- Base type or class (a.k.a. superclass, parent type)
- Derived type or class (a.k.a. subclass, subtype, child type)

Important Aspects

- Programming: implement efficiently a set of related classes (mechanical)
- Design: organize coherently the concepts in an application domain (conceptual)
- Software Engineering: design for flexibility and extensibility in software systems (logical)

```
class Student {
private:
    Name    Nom;
    Address Addr;
    string  Major;
    string  ID;
    int     Level;
public:
    Student(const Name& N, const Address& A, string M = "US",
            string I = "000-00-0000", int L = 10);
    Name    getName() const;
    Student& setName(const Name& N);
    . . .
    string  getMajor() const;
    Student& setMajor(const string& D);
    string  getID() const;
    Student& setID(const string& I);
    int     getLevel() const;
    Student& setLevel(int L);
    ~Student();
};
```

← Specify all the data members

← Specify appropriate constructors

← Specify accessors and mutators for all data members

```
class Employee {
private:
    Name    Nom;
    Address Addr;
    string  Dept;
    string  ID;
public:
    Employee(const Name& N, const Address& A, string D = "",
             string I = "");
    Name    getName() const;
    Employee& setName(const Name& N);
    . . .
    string  getDept() const;
    Employee& setDept(const string& D);
    string  getID() const;
    Employee& setID(const string& I);
    ~Employee();
};
```

← Specify all the data members

← Specify appropriate constructors

← Specify accessors and mutators for all data members

Both classes contain the data members

```
Name      Nom;  
Address   Addr;  
string    ID;
```

and the associated member functions

```
Name      getName() const;  
Address   getAddress() const;  
void      getID() const;  
void      setName(const Name& N);  
void      setAddress (const Address& A);  
void      setID(const string& I);
```

From a coding perspective, this is somewhat wasteful because we must duplicate the declarations and implementations in each class.

From a S/E perspective, this is undesirable since we must effectively maintain two copies of (logically) identical code.

Simply put, we want to exploit the fact that `Student` and `Employee` both are "people".

That is, each shares certain data and function members which logically belong to a more general (more basic) type which we will call a `Person`.

We would prefer to **NOT** duplicate implementation but rather to specify that each of the more specific types will automatically have certain features (data and functions) that are derived from (or inherited from) the general type.

Question: are there any attributes or operations in the overlap that we don't want to include in the base type `Person`?

By employing the C++ inheritance mechanism...

Inheritance in C++ is NOT simple, either syntactically or semantically. We will examine a simple case first (based on the previous discussion) and defer explicit coverage of many specifics until later.

Inheritance in C++ involves specifying in the declaration of one class that it is derived from (or inherits from) another class.

Inheritance may be public or private (or protected). At this time we will consider only public inheritance.

It is also possible for a class to be derived from more than one (unrelated) base class. Such multiple inheritance will be discussed later...

Having identified the common elements shared by both classes (Employee and Student), we specify a suitable base class:

```
class Person {
private:
    Name    Nom;
    Address Addr;
public:
    Person(const Name& N = Name(),
           const Address& A = Address());
    Name    getName() const;
    Person& setName(const Name& N);
    Person& setAddress(const Address& A);
    Address getAddress() const;
    ~Person();
};
```

The base class should contain data members and function members that are general to all the types we will derive from the base class.

Specify public inheritance

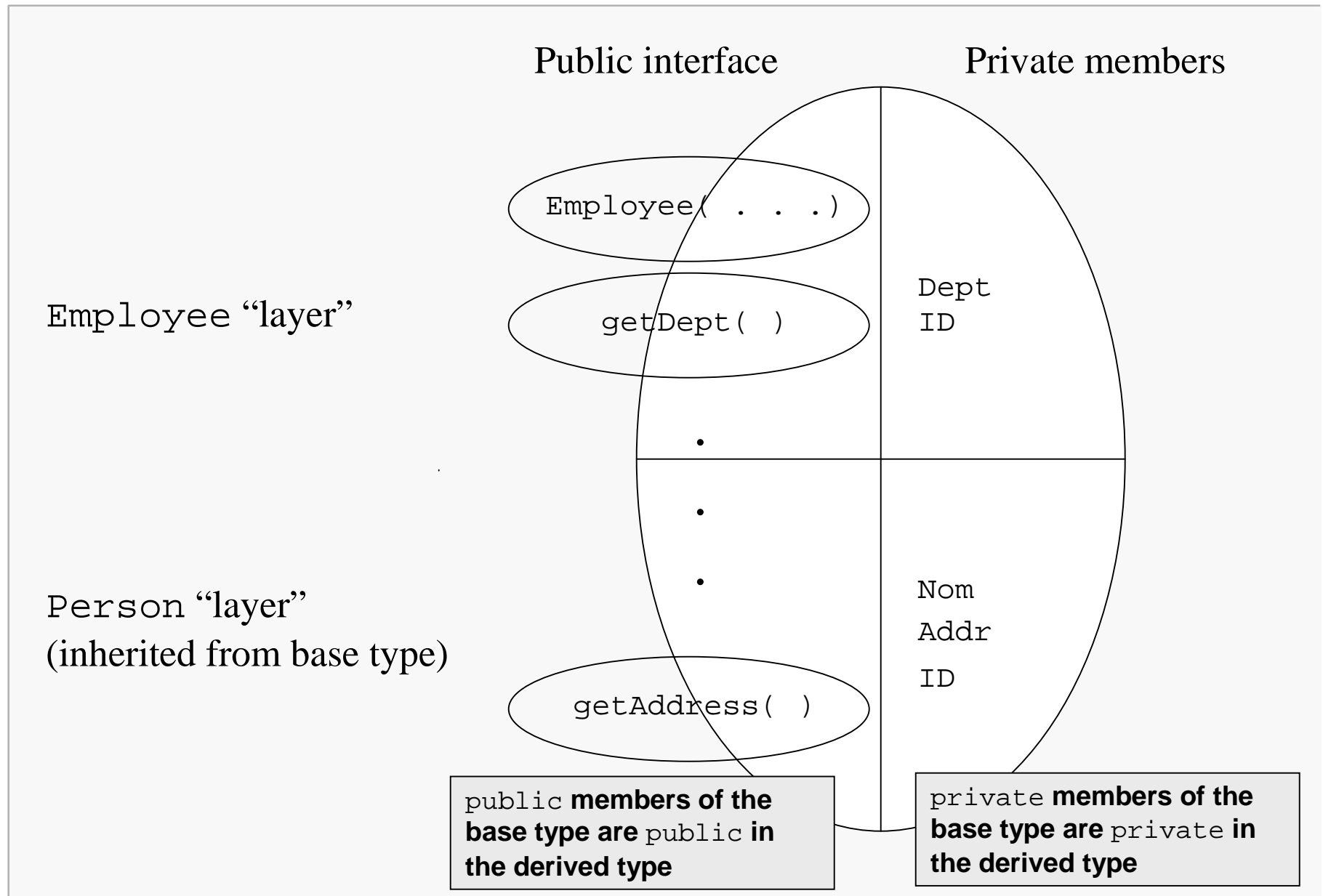
Specify base class

```
class Employee : public Person {  
private:  
    string Dept;  
    string ID;  
  
public:  
    Employee();  
    Employee(const Person& P, const string& D,  
             const string& I);  
    Employee(const Name& N, const Address& A,  
             const string& D, const string& I);  
  
    string getDept() const;  
    Employee& setDept(const string& D);  
    string getID() const;  
    Employee& setID(const string& I);  
  
    ~Employee();  
};
```

Specify additional data members not present in base class

Specify appropriate constructors

Specify accessors and mutators only for the added data members



When an object of a derived type is declared, the default constructor for the base type will be invoked BEFORE the body of the constructor for the derived type is executed (unless an alternative action is specified...).

```
Employee::Employee() : Person() {  
    Dept = "None";  
    ID   = "None";  
}
```

It's not necessary to explicitly invoke the base constructor here, but it makes the behavior more obvious.

Alternatively, the derived type constructor may explicitly invoke a non-default base type constructor:

```
Employee::Employee(const Person& P, const string& D,  
                  const string& I) : Person(P) {  
    Dept = D;  
    ID   = I;  
}
```

Here, the (automatic) copy constructor for the base class is used.

Objects of a derived type inherit the data members and function members of the base type. However, the derived object may not directly access the private members of the base type:

```
Employee::Employee(const Person& P, const string& D,  
                  const string& I) {  
    Nom   = P.getName();  
    Addr  = P.getAddress();  
    Dept  = D;  
    ID    = I;  
}
```

Error: cannot access private member declared in class Person.

For a derived-class constructor we directly invoke a base class constructor, as shown on the previous slide, or use the Person interface:

```
Employee::Employee(const Person& P, const string& D,  
                  const string& I) {  
    setName(P.getName());  
    setAddress(P.getAddress());  
    . . .  
}
```

The restriction on a derived type's access seems to pose a dilemma:

- Having the base type use only public members is certainly unacceptable.
- Having the derived class use the public interface of the base class to access and/or modify private base class data members is clumsy.

C++ provides a middle-ground level of access control that allows derived types to access base members which are still restricted from access by unrelated types.

The keyword `protected` may be used to specify the access restrictions for a class member:

```
class Employee {  
protected:  
    Name      Nom;  
    Address   Addr;  
public:  
    . . .  
};
```



```
Employee::Employee( . . . ) {  
    Nom      = N;           // OK now  
    Addr     = A;  
    Dept     = D;  
    ID       = I;  
}
```

```
class Student : public Person {
private:
    string Major;
    string ID;
    int    Level;

public:
    Student(const Person& P = Person(),
            const string& M = "None",
            const string& I = "000-00-0000", int L = 10);

    string  getMajor() const;
    Student& setMajor(const string& D);
    string  getID() const;
    Student& setID(const string& I);
    int     getLevel() const;
    Student& setLevel(int L);

    ~Student();
};
```

Note that, so far as the language is concerned, Student and Employee enjoy no special relationship as a result of sharing the same base class.

Objects of a derived class may be declared and used in the usual way:

```
. . .
Person JBH(Name("Joe", "Bob", "Hokie"),
           Address("Oak Bridge Apts", "#13", "Blacksburg",
                  "Virginia", "24060"));
```

```
Employee JoeBob(JBH, "Sales", "jbhokie");
```

Call base member

```
cout << "Name:  " << JoeBob.getName().formattedName() << endl
      << "Dept:  " << JoeBob.getDept() << endl
      << "ID:    " << JoeBob.getID() << endl;
```

Base object is only declared to simplify constructor call.

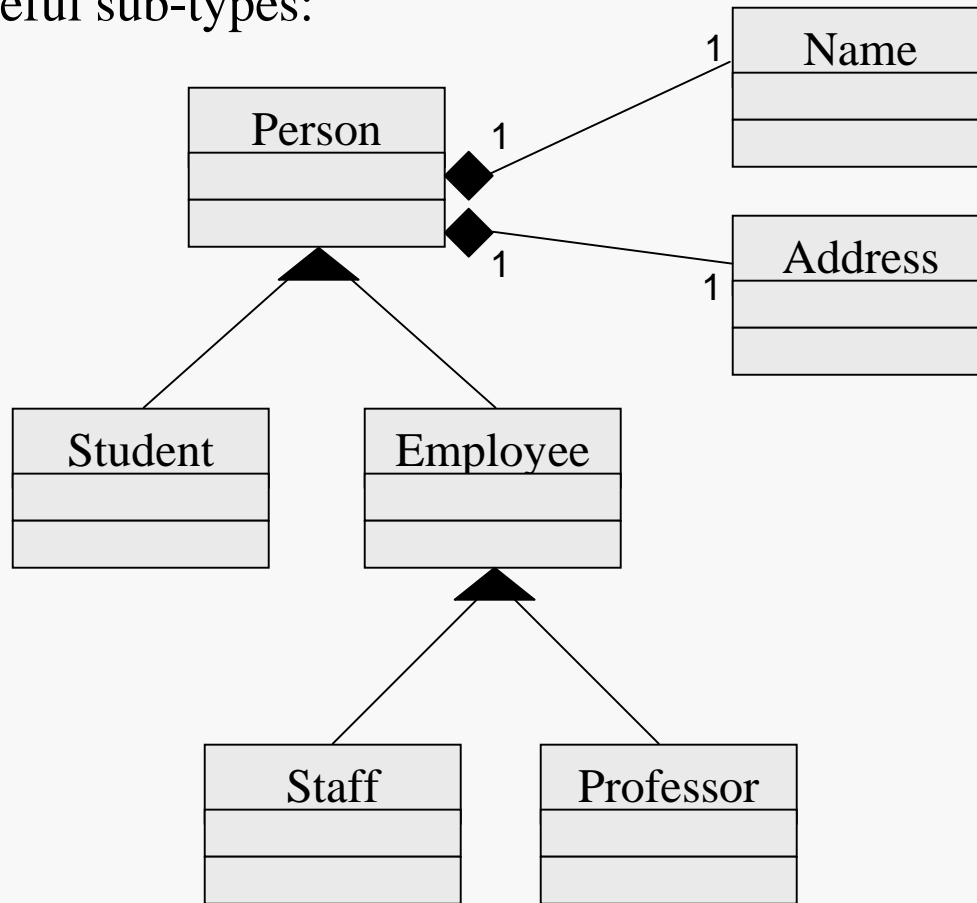
Call derived members

```
Person HHooIV(Name("Haskell", "Horatio", "Hoo"),
              Address("1 Rotunda Circle", "",
                     "Charlottesville", "VA", "21009"));
Student HaskellHoo(HHooIV, "Undecided", "101-01-0101", 40);

HaskellHoo.setAddress(Address("Deke House", "333 Coors Way",
                              "Charlottesville", "VA",
                              "21010"));
HaskellHoo.setMajor("Undeclared");
. . .
```

...to the user there's no evidence here that the class is derived...

Actually, Employee is not a terribly interesting class but it has two (or more) useful sub-types:



There's no restriction on how many levels of inheritance can be designed, nor is there any reason we can't mix inheritance with association and/or aggregation.

For the sake of an example, a staff member is paid an hourly wage, so the class `Staff` must provide the appropriate extensions...

```
class Staff : public Employee {
private:
    double HourlyRate;
public:
    Staff(const Employee& E, double R = 0.0);
    double getRate() const;
    void    setRate(double R);
    double grossPay(int Hours) const;
    ~Staff();
};
```

...whereas a professor is paid a fixed salary:

```
class Professor : public Employee {
private:
    double Salary;
public:
    Professor(const Employee& E, double S = 0.0);
    double getSalary() const;
    void    setSalary(double S);
    double grossPay(int Days) const;
    ~Professor();
};
```

The base member function `Employee::setID()` is simple:

```
Employee& Employee::setID(const string& S) {  
    ID = S;  
    return (*this);  
};
```

This implementation raises two issues we should consider:

- What if there's a specialized way to set the ID field for a derived type?
- Is the return type really acceptable for a derived type?

We'll consider the first question now... suppose that the ID for a professor must begin with the first character of that person's department.

Then `Professor::setID()` must enforce that restriction.

In the derived class, provide an appropriate implementation, using the same interface. That will override the base class version when invoked on an object of the derived type:

```
Professor& Professor ::setID(const string& I) {  
    if (I[0] == Dept[0])  
        ID = I;  
    else  
        ID = Dept[0] + I;  
    return (*this);  
};
```

The appropriate member function implementation is chosen (at compile time), based upon the type of the invoking object and the inheritance hierarchy. Beginning with the derived class, the hierarchy is searched upward until a matching function definition is found:


```
Employee E(. . .);  
Professor F(. . .);  
.  
.  
.  
E.setID("12334"); // Employee::setID()  
F.setID("99012"); // Professor::setID()
```

Suppose we added a display member function to the base type:

```
void Person::Display(ostream& Out) {  
    Out << "Name:    " << Name << endl  
        << "Address: " << Address;  
};
```

This is inadequate for a Professor object since it doesn't recognize the additional data members... we can fix that by overriding again (with a twist):

```
void Professor::Display(ostream& Out) {  
    Person::Display(Out);  
    Out << "ID:      " << ID << endl  
        << "Dept:   " << Department;  
};
```



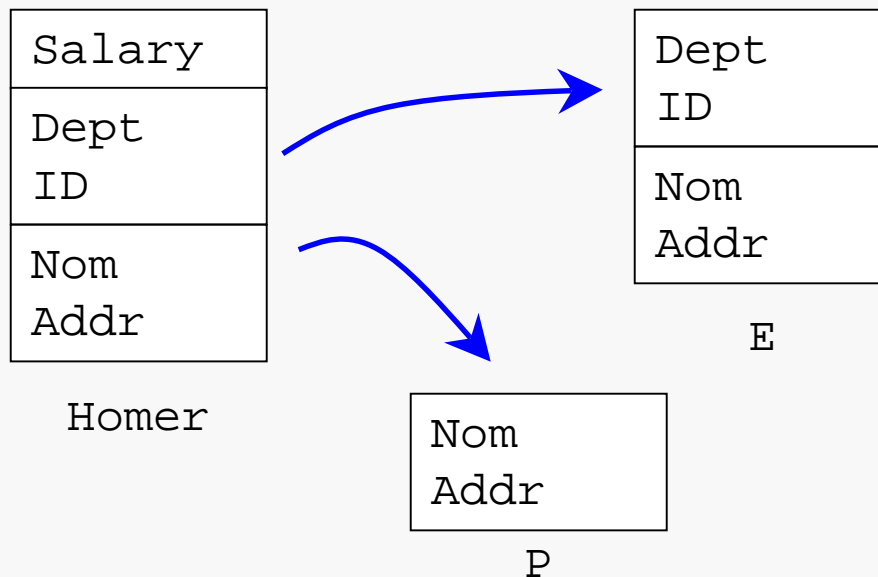
Here, we use the base class display function, invoking it with the appropriate scope resolution, and then extend that implementation with the necessary additional code.

It is legal to assign a derived type object to a base type object:

```
Employee eHomer(Name("Homer", "P", "Simpson"),
                Address("1 Chernenko Way", "", "Blacksburg",
                       "VA", "24060"),
                "Physics", "P401" );
Professor Homer(eHomer, 45000.00);

Employee E;
Person P;

E = Homer; // legal assignments, but usually inadvisable
P = Homer;
```



When a derived object is assigned to a base target, only the data and function members appropriate to the target type are copied.

This is known as slicing.

```
void PrintEmployee(Employee toPrint, ostream& Out) {  
    Out << toPrint.getID();  
    Out << '\t';  
    Out << toPrint.getName();  
    Out << '\n';  
}
```

`PrintEmployee()` sees only the `Employee` layer of the actual parameter that was passed to it by value.

That's actually OK in this case since that's all `PrintEmployee()` deals with anyway.

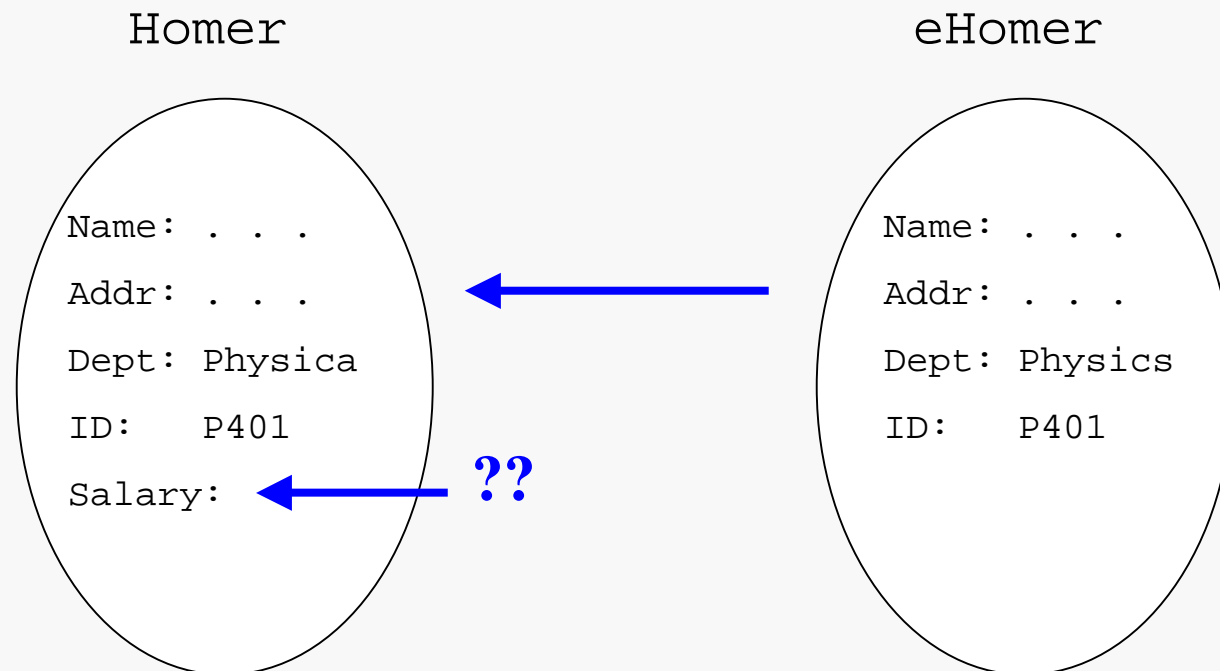
However, it's certainly a limitation you must be aware of... what if you wanted to write a generic print function that would accept any derived type, without slicing?

By default, a base type object may not be assigned to a derived type object:

```
// assume declarations from slide 24. . .  
Homer = eHomer;    // illegal - compile time error
```

It's possible to legalize this with the right overloading (later), but...

... some sort of action must be taken with respect to the derived type data members that have no analogs in the base type.



The rules are essentially the same in four situations:

- when passing a derived object by value as a parameter.
- when returning a derived object by value
- when initializing a base object with a derived object
- when assigning a derived object to a base object

A derived type may be copied when a base type is targeted — however, slicing will occur.

A base type may not be copied when a derived type is targeted — unless a suitable derived type copy constructor is provided to legalize the conversion.