DigiPen INSTITUTE OF TECHNOLOGY

CS170#12.1

Polymorphism

Vadim Surov

Outline

- Definition
- Example Of A Class Hierarchy
- Slicing Effect
- C++ Support For Polymorphism
- Early Binding
- Late Binding

Polymorphism

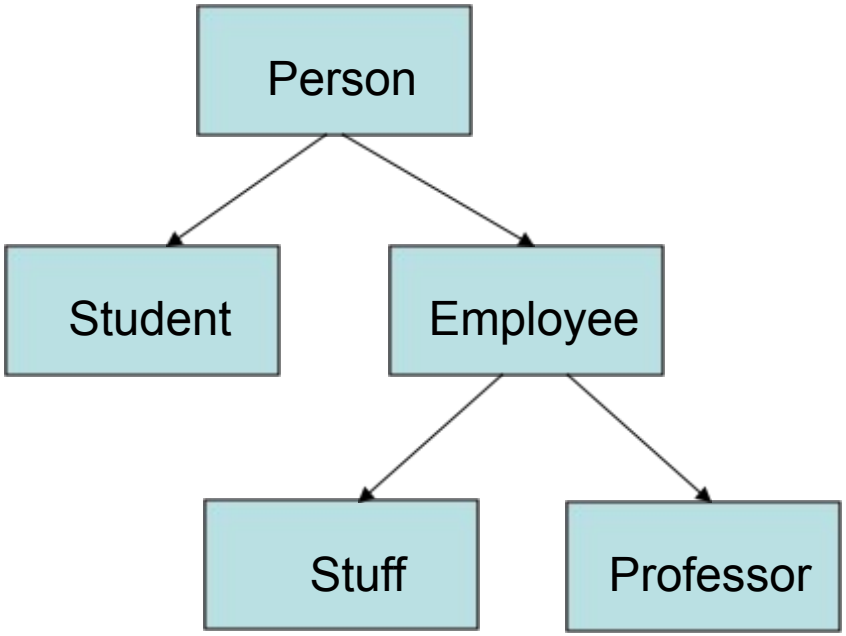
- OOP pillars: encapsulation, inheritance and **polymorphism**

- **Polymorphism - ability to obtain type-specific behavior based on the dynamic type of a reference or pointer**

- A term derived from a Greek word that means “many forms”

Example Of A Class Hierarchy

- Let's assume we have this class hierarchy

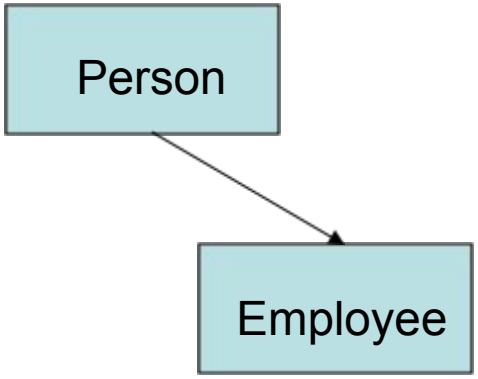


Example Of A Class Hierarchy (contd)

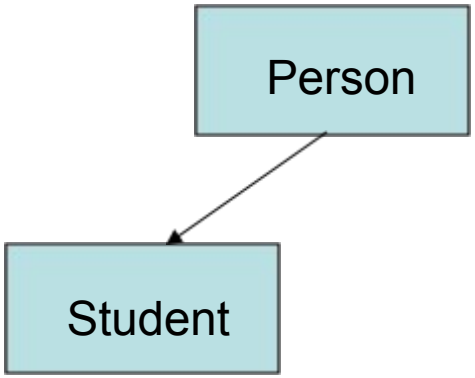
```
class Person {

private:
    Name name;
    Address address;

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();
};
```

Example Of A Class Hierarchy (contd)

```
class Employee : public Person {
private:
    string Department;
    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 getDepartmeny() const;
    Employee& setDepartment(const string& D);
    string getID() const;
    Employee& setID(const string& I);
    ~Employee();
};
```



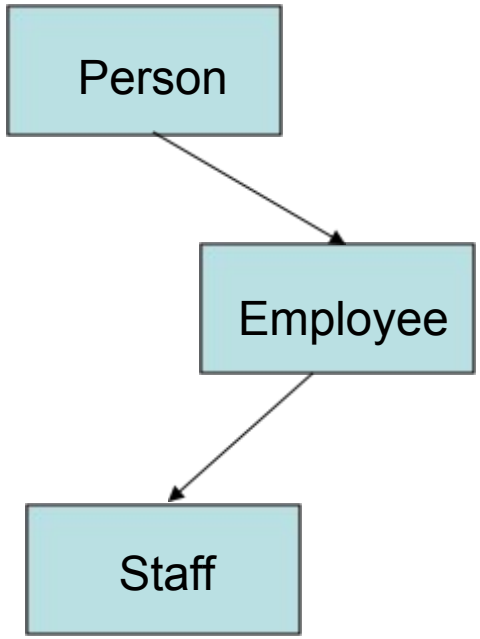
Example Of A Class Hierarchy (contd)




```
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();
};
```

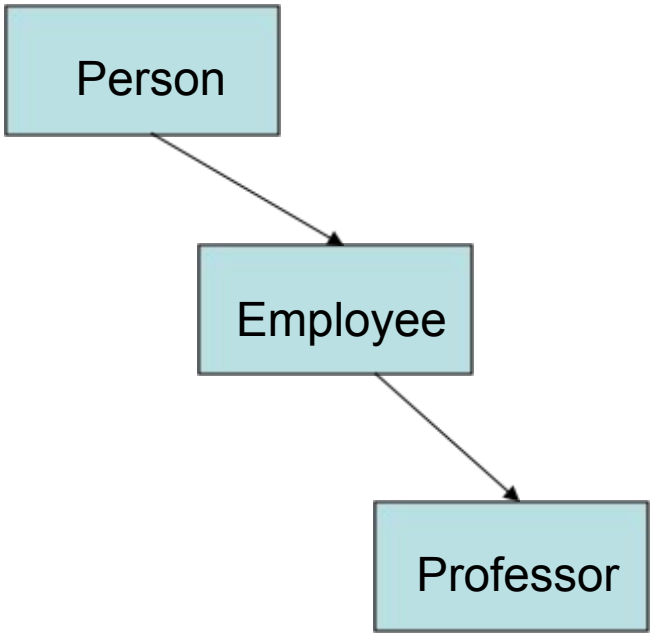
Example Of A Class Hierarchy (contd)

```
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();
};
```



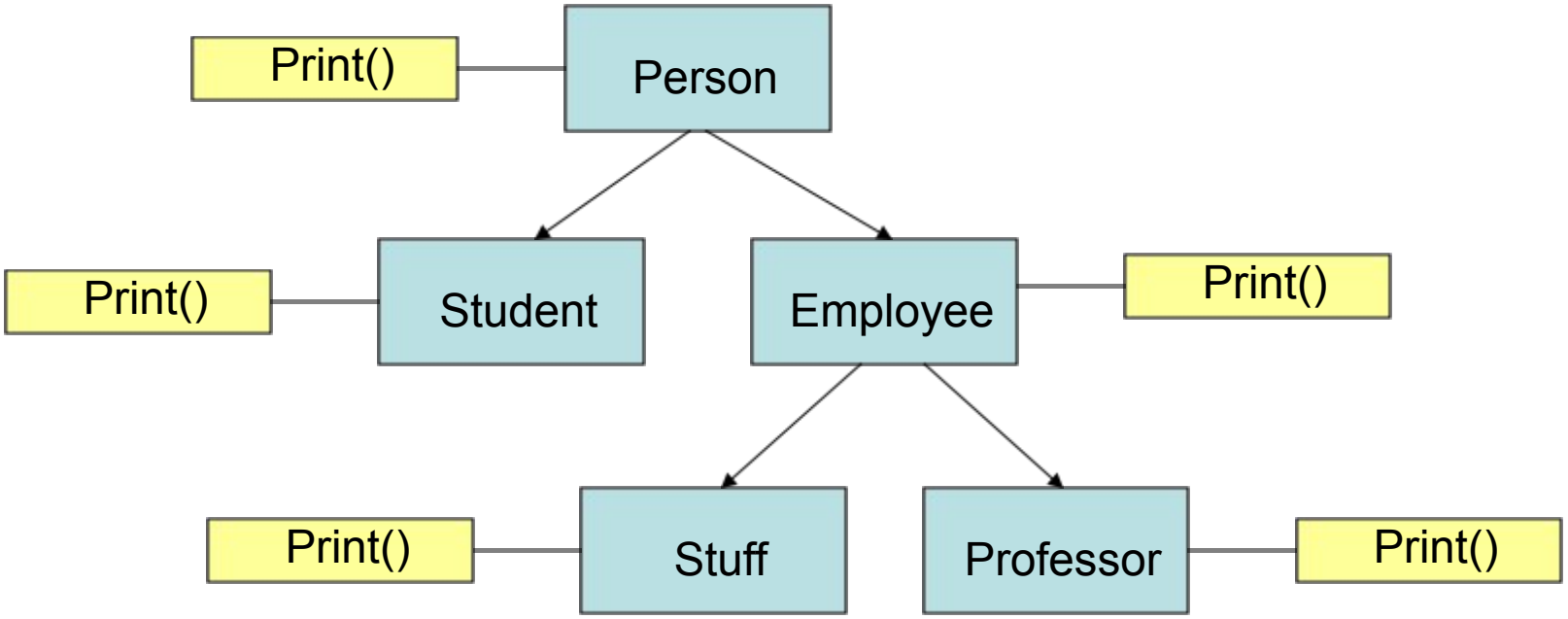
Example Of A Class Hierarchy (contd)

```
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();
};
```



Example Of A Class Hierarchy (contd)

- Assume that a member function `Print()` has been added to `Person`, and overridden with custom versions in `Student`, `Staff` and `Professor`



Example Of A Class Hierarchy (contd)

- The base `print` function couts `Name` and `Address`:

```
void Person::Print() {
   cout << "Name: " << name << endl;
   cout << "Address: "<< address << endl;
}
```

- `Professor::Print()` invokes the base function and extends it:

```
void Professor::Print() {
   Person::Print();
   cout << "Department: " << getDeptartment() << endl;
   cout << "ID: " << getID() << endl;
   cout << "Salary: " << Salary << endl;
}
```

- `Student::Print()` and `Staff::Print()` are similar

Organizing Objects Of Related Types

- It is somewhat reasonable to want to have a single data structure that holds objects of any type derived from `Person`
- We may achieve this by using an array, declared to hold objects of the base type `Person`:

```
Professor JoeBob();
Student HaskellHoo();
Staff JillAnne();
Person People[3];
People[0] = JoeBob;
People[1] = HaskellHoo;
People[2] = JillAnne;
```

- Since it is legal to assign a derived type object to an object of its base type, the storage statements here are legal ...

Slicing Effect

- ... however the assignment of a derived object to a base variable results in "slicing": **the non-base elements are lost in the copying**
- As result, the three invocations of `Print()` below

```
Person People[3];
People[0] = JoeBob;
People[1] = HaskellHoo;
People[2] = JillAnne;
People[0].Print();
People[1].Print();
People[2].Print();
```

act on objects of type `Person`, and the resulting output shows only the data elements of a `Person` object

Using Pointers To Avoid Slicing

- So this result is not acceptable. We want to use a single data structure to hold objects of related but different types, and have the resulting behavior reflect the type of the actual object

- Q: How can we avoid slicing?
- A: **Don't make a copy!** If our data structure stores pointers to the objects, then no slicing will occur

- But is this legal? And what is the effect now if we access the objects?

Pointer Access To Objects In A Hierarchy

```
Professor JoeBob();
Student HaskellHoo();
Staff JillAnne();

Person* People[3];

People[0] = &JoeBob;
People[1] = &HaskellHoo;
People[2] = &JillAnne;



People[0]->Print();
```

- The code just shown is legal. A base-type pointer may store the address of a derived-type object
- The effect, however, is no better than before. The following statement will still invoke `Person::Print()` and display only the data members that belong to the `Person` layer of the object

Pointer Access To Objects In A Hierarchy (contd)

```
Professor JoeBob();
Student HaskellHoo();
Staff JillAnne();

Person* People[3];

People[0] = &JoeBob;
People[1] = &HaskellHoo;
People[2] = &JillAnne;



People[0]->Print();
```

- The base pointer does point to an object of type `Professor`, but the derived layer with its extended functionality is still inaccessible
- So what we have to do more to achieve this polymorphic behaviour?

C++ Support For Polymorphism

- In C++, polymorphic behavior can be achieved by combining:
  - an inheritance hierarchy
  - object accesses via pointers or references
  - use of **virtual** member functions in base classes

C++ Support For Polymorphism (contd)

- A member function is declared virtual by simply preceding its prototype with the keyword `virtual`

```
class Person {
public:
  Person();
  virtual void Print();
};
```

- By declaring `Person::Print()` as `virtual`, we complete the enabling of the mechanism used in C++ to achieve polymorphic behavior

Virtual Method Visibility

- Guideline: make virtual functions **private** if it exists only to achieve polymorphic behavior
- If they also need to be invoked directly from within derived classes' code, keep them **protected**
- Otherwise, keep them **public**

```
class Person {
public:
  Person();
private:
  virtual void Print();
};
```

Virtual Functions And Binding

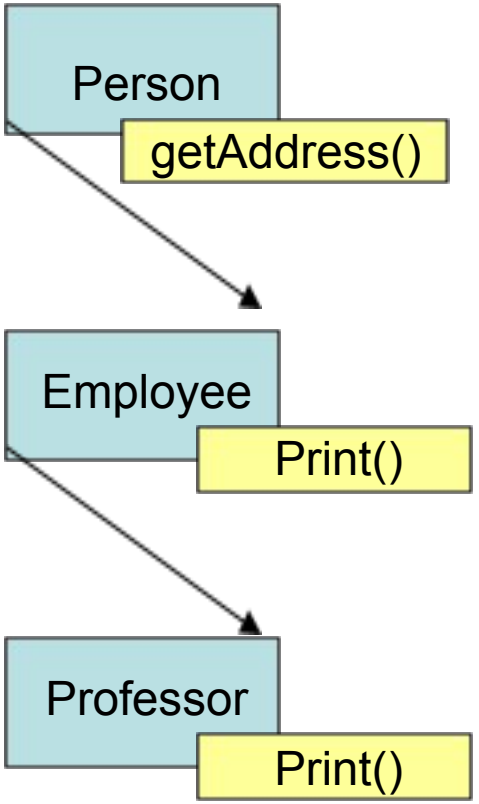
- Normally functions are declared virtual in a base class and then overridden in each derived class for which the function should have a specialized implementation
- In normal circumstances the compiler determines how to bind each function call to a specific implementation by searching within the current scope for a function whose signature matches the call, and then expanding that search to enclosing scopes if necessary
- With an inheritance hierarchy, that expansion involves moving back up through the inheritance tree (to base) until a matching function implementation is found

Early Binding

- When the binding of call to implementation takes place at compile-time we say we have **early binding** (AKA static binding)
- Following code

```
Professor p();
p.Print();
cout << p.getAddress();
```

binds to the local implementation of `Print()` given in the class `Professor`, which overrides the one inherited from the base class `Employee`, and this binds to the implementation of `getAddress()` inherited from the class `Person`



Early Binding (contd)

- Early binding is always used if the invocation is direct (using the dot operator), whether virtual functions are used or not:

```
Professor JoeBob();

Person* people[3];
people[0] = &JoeBob;

people[0]->Print();    // May be an early binding
(*people[0]).Print(); // Early binding
```

Invocation Via A Pointer Without Virtuality

- When a function call is made using a pointer, and no virtual functions are involved, the binding of the call to an implementation is based upon the type of the pointer (not the actual type of its target)

- In this example, the second call binds to the local implementation of `setID()` given in the class `Employee`

```
Professor p();
p.setID();      // 1
Employee* pEmp = &p;
pEmp->setID(); // 2
Person* pPer = &p;
pPer->setID(); // 3
```

- The last call produces a compile-time error because the class Person does not provide a member function that matches the call

Enabling Polymorphism With Virtual Functions

- However, when a function call is made using a pointer, and virtual functions are involved, the binding of the call to an implementation is based upon the type of the target object (not the declared type of the pointer)

- Let's modify the declaration of `Employee` to make `setID()` a virtual function

```
class Employee :
    public Person {
private:
   string Dept;
   string ID;
public:
   virtual Employee&
      setID(const string& I);
   ~Employee();
};
```

Invocation Via A Pointer With Virtuality

- Now, if we access objects in this inheritance hierarchy via pointers, we get polymorphic behavior
- This call now

```
Professor p();
p.setID();

Employee* pEmp = &p;
pEmp->setID();
```

binds to the overriding implementation of `setID()` given in the class `Professor`, because `*pEmp` is an object of that type

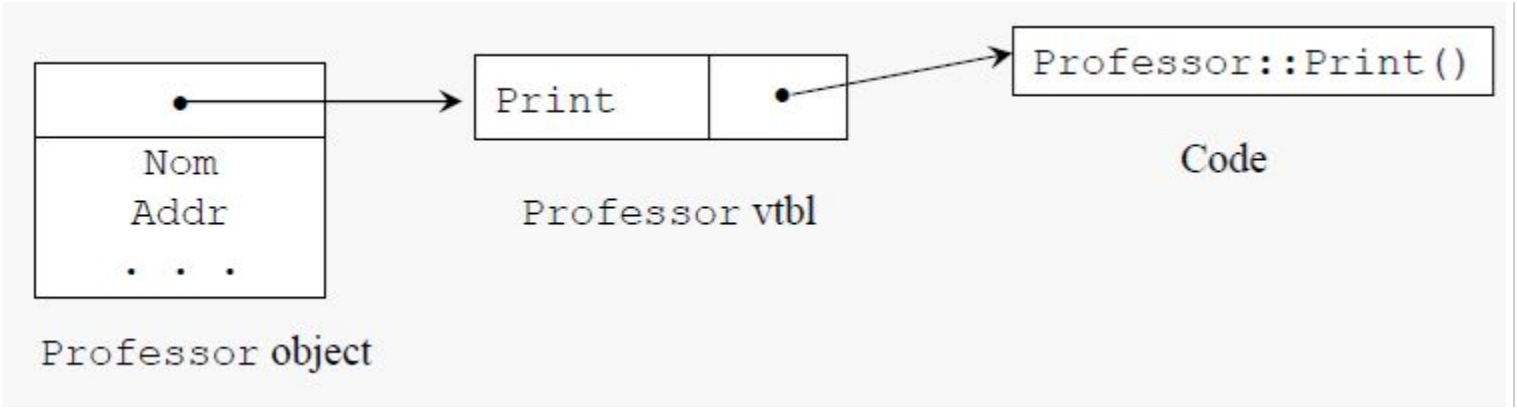
Late Binding

- When the binding of call to implementation takes place at runtime we say we have **late binding** (AKA dynamic binding)
- There's no way to know the type of the target of `pPer` until runtime
- During runtime the call to the virtual member function `Print()` will be bound to the correct implementation

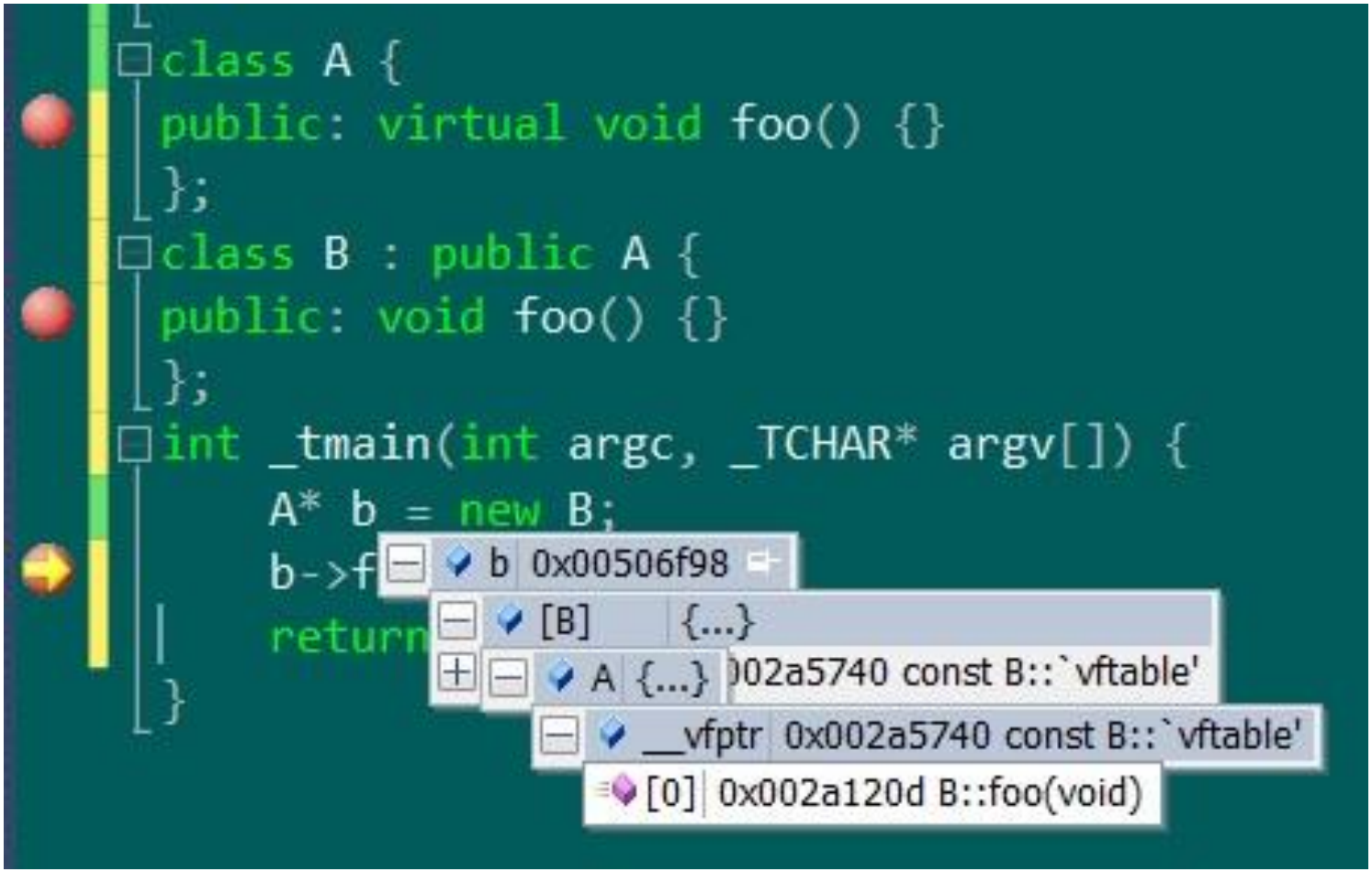
```
Professor prof();
Staff staff;

Person* pPer;
char ch;
cout << "Enter choice: ";
cin >> ch;
if (ch == 'y')
   pPer = &prof;
else
   pPer = &staff;
pPer->Print();
```

Virtual Function Tables

- When the binding of call to implementation takes place at runtime, the address of the called function must be managed dynamically
- The presence of a virtual function in a class causes the generation of a **virtual function table** for the class



Virtual Function Tables



Base Class Shortcomings

- In the following code:

```
Professor JoeBob();
Person* pPer = &JoeBob;

pPer->setID("P0007"); // Illegal
```

the attempt to call `Professor::setID()` is illegal because the class that corresponds to the pointer type (`Person`) doesn't have a matching function

Base Class Shortcomings (contd)

- We could fix this by adding a corresponding virtual function to the base type `Person`, but such a function doesn't make any sense since `Person` doesn't store an `ID` string
- Nevertheless, designers often resort to clumsy fixes like this to make a hierarchy work
- If we add `Person::setID()` as a virtual protected function, it is invisible to `Person` clients but can still be overridden in derived classes
- Unfortunately that would not fix the access problem in the code above (it's protected!)

Pure Virtual Functions

- The problem here is somewhat nasty…we can add an unsuitable public member function to the base class, or we can resort to placing a **pure virtual function** in the base class:

```
class Person {
private:
    Name Nom;
    Address Addr;
public:
    virtual Person& setID(const string& I) = 0;
    virtual void Print(ostream& Out);
    ...
};
```

Pure Virtual Functions (contd)

- A pure virtual function can be not implemented in the base class, but must be implemented in a derived class
- If not implemented in the base, this means it is no longer possible to declare an object of type `Person`
- A class that cannot be instantiated is called an **abstract class**

A Design Lesson

- We will adopt this approach, making `Person` an abstract base class for the entire hierarchy
- This has some costs… a number of the member functions in the derived classes (chiefly constructors) must be revised because it is now impossible to declare an object of type `Person`, even anonymously
- We would have been much better off if this decision had been made early, before so many derived classes were implemented
- On the other hand, it is very common to have an abstract base class, usually because in the end that base class turns out to be so general (or so problematic) that we will never instantiate it

The Revised Person Class

```
class Person {
private:
   Name Nom;
   Address Addr;
public:
   Person(const Name& N = Name(),
          const Address& A = Address());
   Person& setAddress(const Address& newAddr);
   Address getAddress() const;
   Name getName() const;
   Person& setName(const Name& N);
   virtual Person& setID(const string& I) = 0;
   virtual string getID() const = 0;
   virtual void Print(ostream& Out);
   virtual ~Person();
};
```

The Revised Employee Class

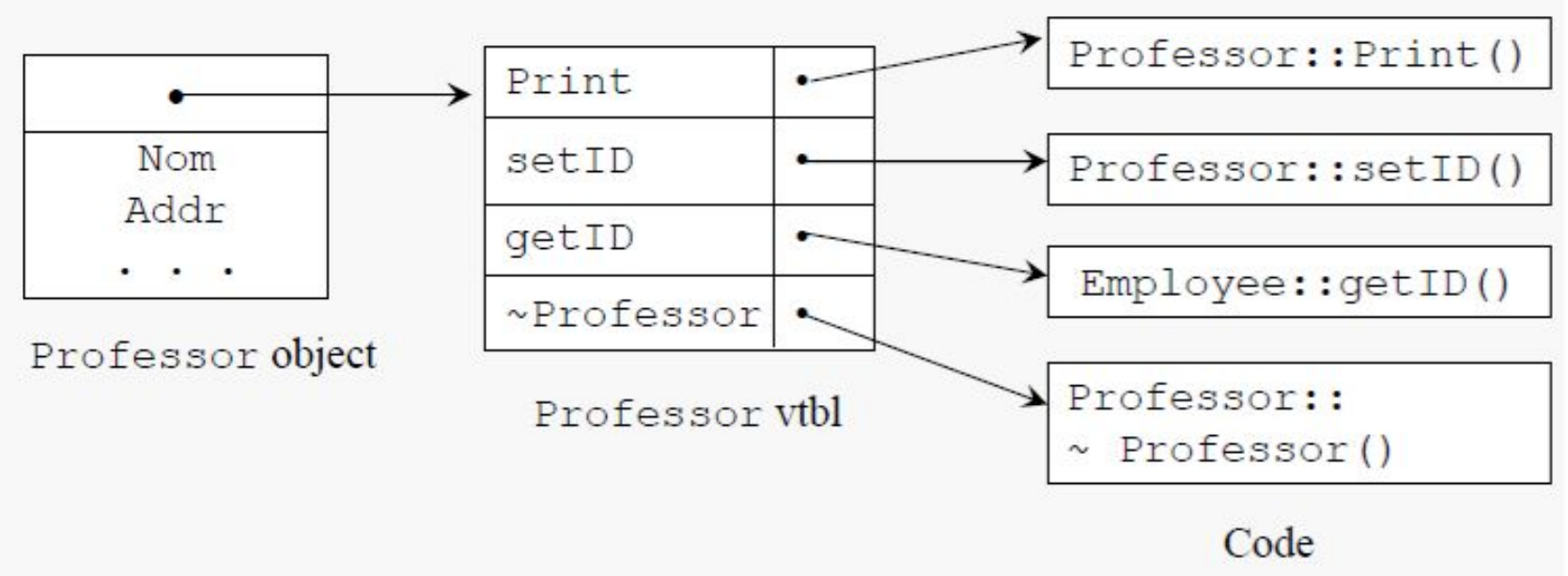
```
class Employee : public Person {
private:
   string Dept;
   string ID;
public:
   Employee();
   Employee(const Name& N, const Address& A,
            const string& D, const string& I);
   string getDept() const;
   Employee& setDept(const string& D);
   virtual string getID() const;
   virtual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

The Revised Professor Class

```
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;
   virtual Person& setID(const string& I);
   virtual void Print(ostream& Out);
   ~Professor();
};
```

Late Binding Revisited

- At runtime, the `Professor` vtbl structure looks something like this:



So, At Runtime

```
Person* pPer;
Professor* pProf = new Professor();
pPer = pProf;
pPer->setID();
cout << pPer->getID();
```

So, At Runtime (contd)