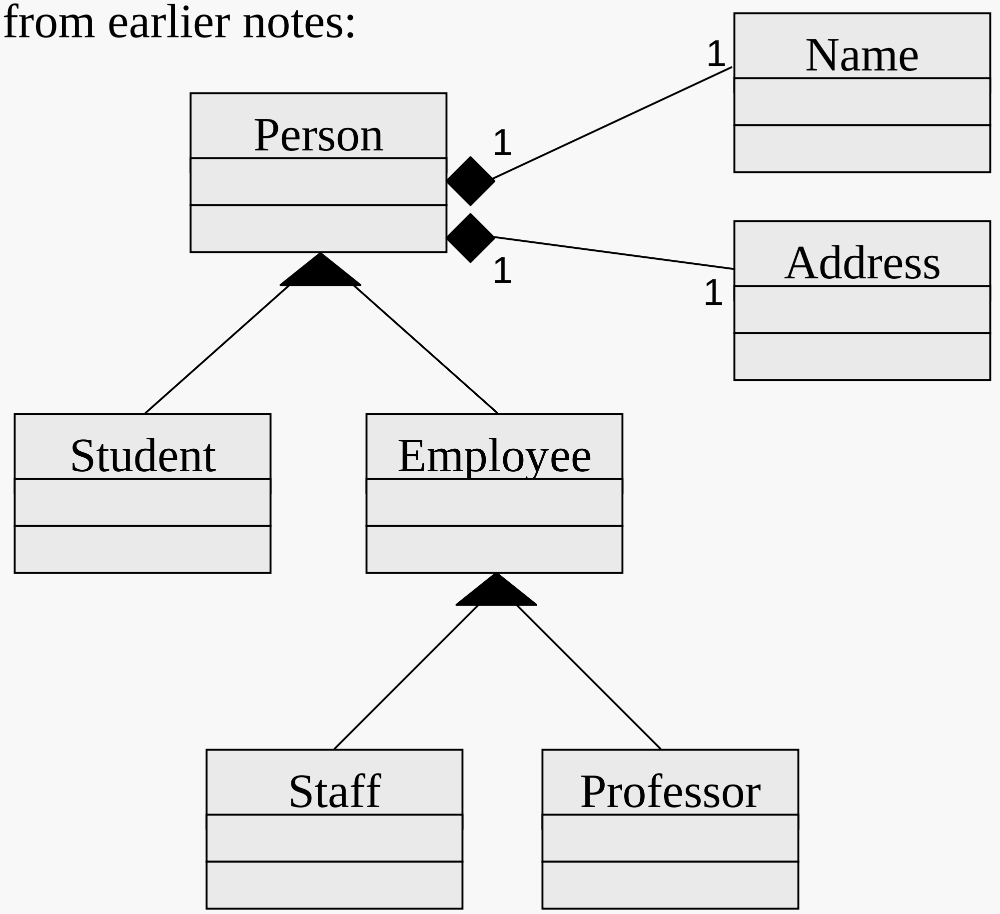
Definition

polymorphism	the ability to manipulate objects of distinct classes using only knowledge of their common properties without regard for their exact class (Kafura)

Note that polymorphism involves both algorithms and types of data (classes).

Class Hierarchy

Recall the inheritance hierarchy from earlier notes:



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

Base Function and Specializations

The base print function uses an overloading of `operator<<` for the class `Address`:

```
void Person::Print(ostream& Out) {

   Out << "Name:   " << Nom.formattedName() << endl
       << Addr << endl;
}
```

```
void Professor::Print(ostream& Out) {

   Person::Print(Out);
   Out << "Dept:   " << getDept() << endl;
   Out << "ID:     " << getID() << endl;
   Out << "Salary: " << fixed << showpoint
       << setw(10) << setprecision(2) << Salary << endl;
}
```

`Professor::Print()` invokes the base function and extends it.
`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`. For example:

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

Person People[3];
People[0] = JoeBob;
People[1] = HaskellHoo;
People[2] = JillAnne;
```

We may achieve this by using a structure, such as an array, declared to hold objects of the base type, `Person` in this case.

Since it is legal to assign a derived type object to an object of its base type, the storage statements here are legal… however, the effect is not ideal…

Slicing Again

As noted earlier, the assignment of a derived object to a base variable results in "slicing"; the non-base elements are lost in the copying. Thus:

```
Person People[3];
People[0] = JoeBob;
People[1] = HaskellHoo;
People[2] = JillAnne;

People[0].Print(cout);
People[1].Print(cout);
People[2].Print(cout);
```

The three invocations of `Print()` act on objects of type `Person`, and the resulting output shows only the data elements of a `Person` object.

Of course, you can't expect anything better because the elements of the array `People[]` are simply objects of type `Person`.

Using Pointers to Avoid Slicing

Actually the results are not really acceptable. There are times when we definitely want to be able 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.

The first question: is how can we avoid slicing?

The answer is: don’t make a copy…

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

Person* People[3];
People[0] = &JoeBob;
People[1] = &HaskellHoo;
People[2] = &JillAnne;
```

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

The code just shown is this 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.

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

The base pointer does point to an object of type `Professor` (see the original declaration and assignment), but the derived layer with its extended functionality is still inaccessible…

… but this illustrates two of the three steps that are necessary to achieve our goal.

C++ Support for Polymorphism

In C++, polymorphic behavior can be attained by combining:

- an inheritance hierarchy
- object accesses via pointers
- use of virtual member functions in base classes

```
class Person {
 . . .
public:
   Person(. . .);
   . . .
   virtual void Print(ostream& Out);
};
```

```
Professor JoeBob( . . . );
. . .
Person* People[3];
People[0] = &JoeBob;
. . .
People[0]->Print(cout);
```

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

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

Virtual Functions and Binding

A member function is declared to be virtual by using the keyword `virtual`.

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.

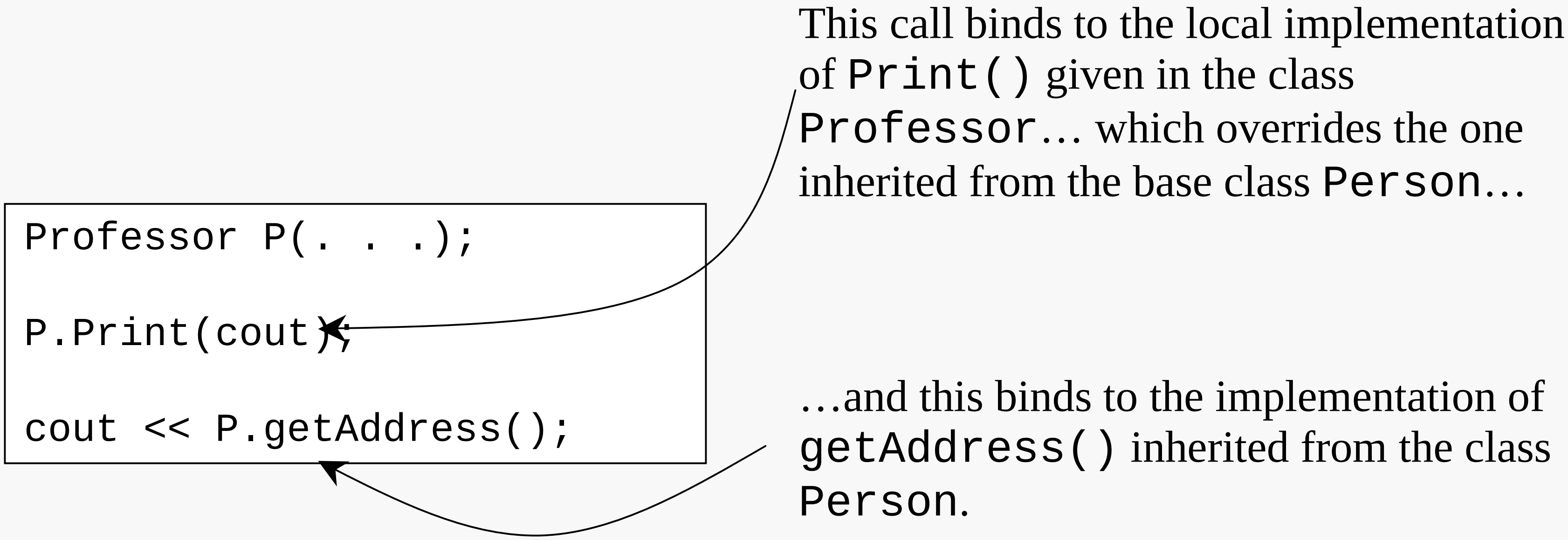
This modifies the rules whereby a function call is bound to a specific function implementation.

In normal circumstances (i.e., what we've done before) 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 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).



```
Professor P(. . .);

P.Print(cout);

cout << P.getAddress();
```

This call binds to the local implementation of `Print()` given in the class `Professor`… which overrides the one inherited from the base class `Person`…

…and this binds to the implementation of `getAddress()` inherited from the class `Person`.

Early binding is always used if the invocation is direct (via the name of an object using the dot operator), whether virtual functions are used or not.

The search for a matching function begins with the actual object type and proceeds up the inheritance hierarchy until a match is found.

Invocation via a Pointer w/o 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).

```
Professor P(. . .);
P.setID(. . .);


Employee* pEmp = &P;
pEmp->setID(. . .);


Person* pPer = &p;
pPer->setID(. . .);
```

This call binds to the local implementation of `setID()` given in the class `Employee`.

This call would bind to the implementation of `setID()` inherited from the class `Person`, if there were one. As it is, this will generate an error at compile time.

Note: 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).

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();
};
```

Note this doesn't change the implementation of `Employee::setID()`.

Invocation via a Pointer with Virtuality

Now, if we access objects in this inheritance hierarchy via pointers, we get polymorphic behavior. That is, the results are consistent with the type of the target, rather than the type of the pointer:

```
Professor P(. . .);
P.setID(. . .);


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

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

If you don’t think that’s cool...

The point is that we trigger the behavior of the actual object, even though we can't tell what that type is from the invocation.

Late Binding

When the binding of call to implementation takes place at runtime we say we have <u>late binding</u> (aka <u>dynamic</u> binding).

```
Professor Prof(. . .);
Staff     Stff(. . .);

Person* pPer;

char ch;
cout << "Enter choice: ";
cin >> ch;
if (ch == 'y')
   pPer = &Prof;
else
   pPer = &Stff;

pPer->Print(cout);
```

There’s no way to know the type of the target of `pPer` until runtime.

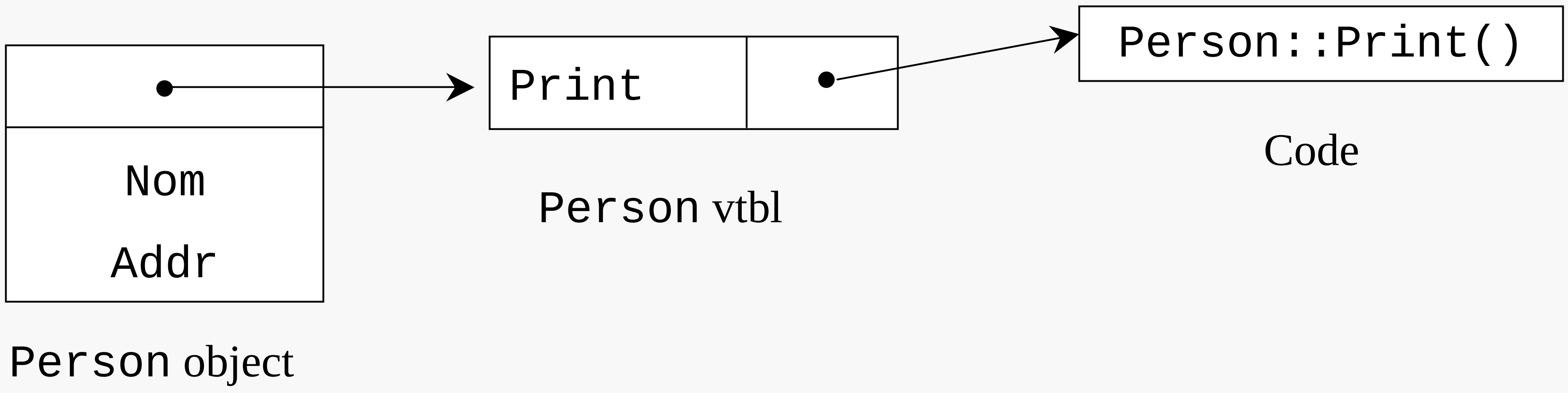
However, the call to the virtual member function `Print()` will be bound to the correct implementation regardless.

But HOW is this done? See Stroustrup 2.5.5 and 12.2.6.

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 (vtbl) for the class, and an association to that table in each object of that type:

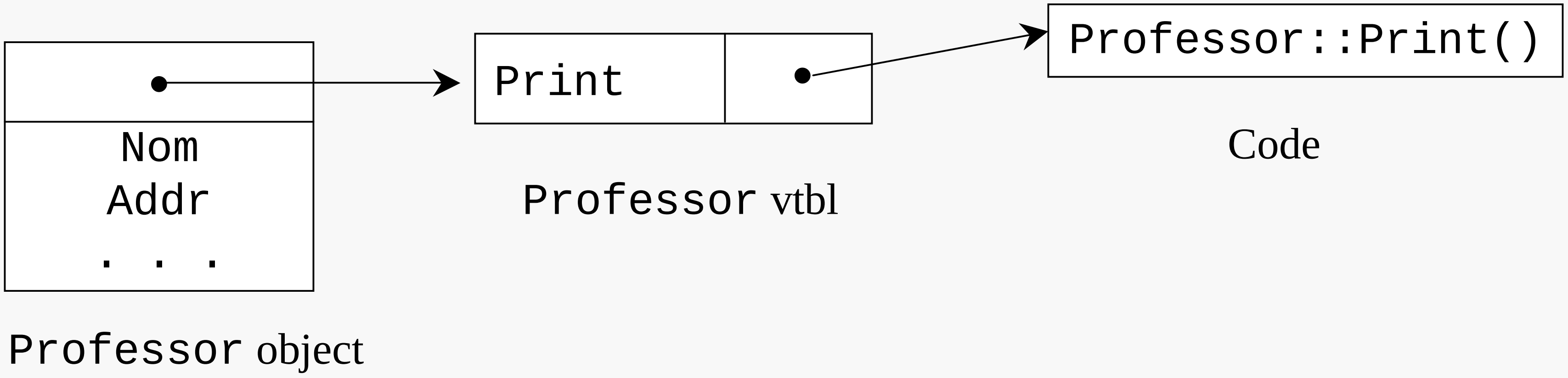


This increases the size of each object, but only by the size of one pointer.

Key fact: if a function is virtual in a base class, it's also virtual in the derived class, whether it's declared virtual there or not.

Derived Class View

So for a `Professor` object we'd have:



`Nom`
`Addr`
`. . .`

`Professor` object

`Print`

`Professor` vtbl

`Professor::Print()`

Code

In this simple case, the derived object has its own implementation to replace the single virtual function inherited from the base class.

That's often NOT the case. Then, one or more of the derived class vtbl pointers will target the base class implementation… but first we have another problem to address.

Base Class Shortcomings

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

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

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

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.

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);
   . . .
};
```

A pure virtual function does not have an implementation.

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

Here is our revised base 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();
};
```

Note that the constructor is retained because it is useful in the derived type constructors.

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();
};
```

All the member functions will be implemented, so `Employee` is not an abstract class.

If we did not implement the inherited pure virtual functions, `Employee` would still be abstract.

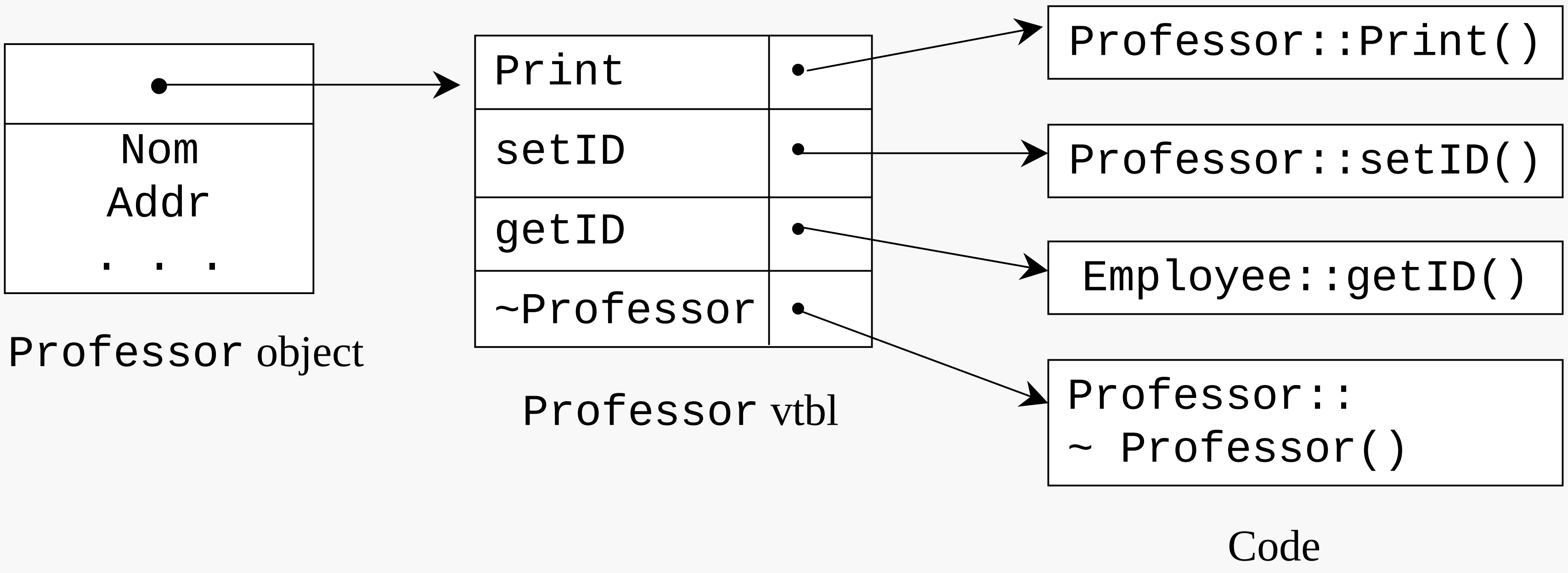
The Revised `Professor` class

```
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();
};
```

```
Person* pPer;
Professor* pProf = new Professor(. . .);

pPer = pProf;
pPer->setID(. . .);
cout << pPer->getID();
```

Late Binding Revisited

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

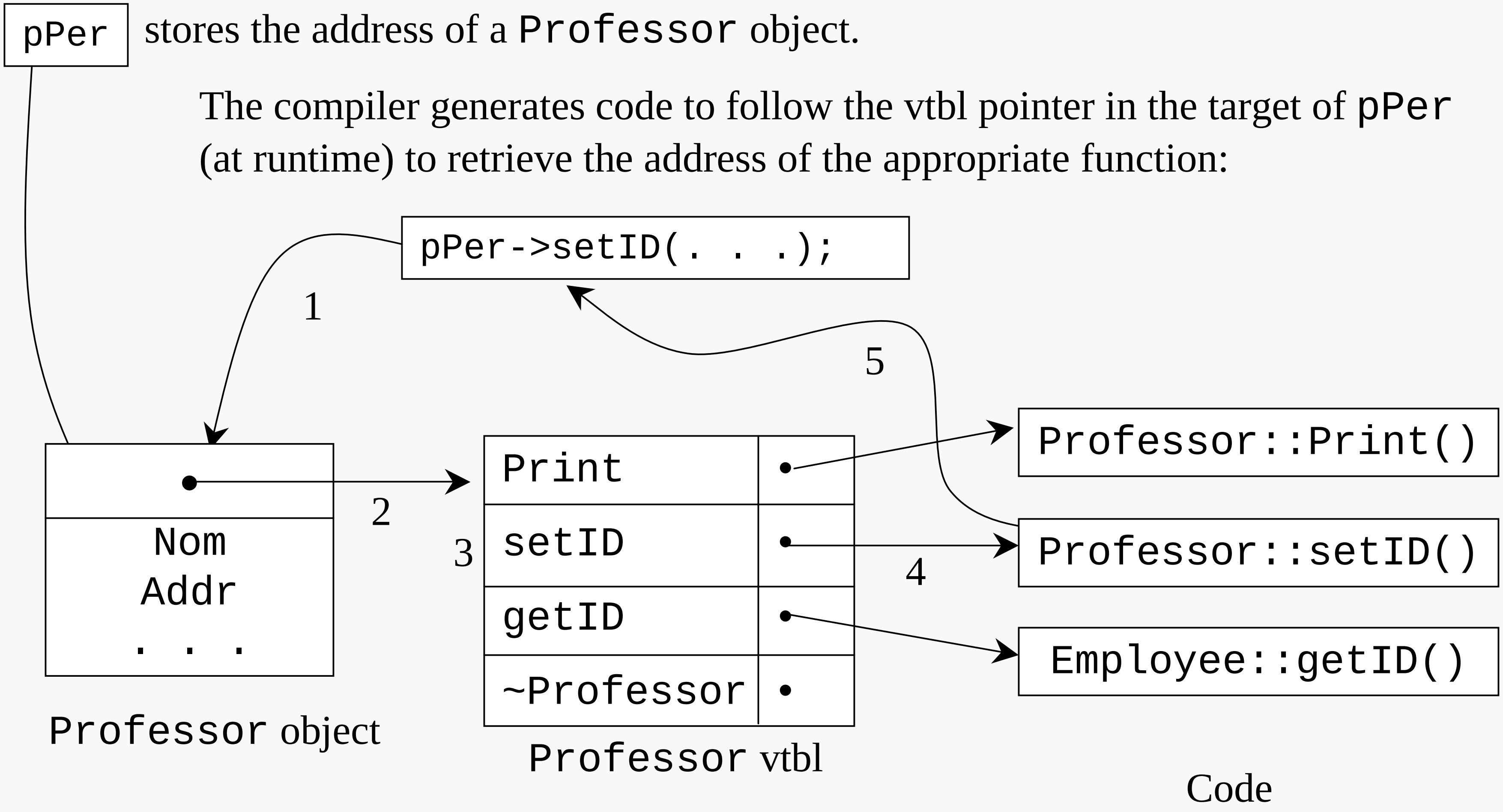


So, at Runtime:

Resuming the example from slide 22:

`pPer` stores the address of a `Professor` object.

The compiler generates code to follow the vtbl pointer in the target of `pPer` (at runtime) to retrieve the address of the appropriate function:



Detail: the vtbl pointer must be at a fixed offset within the target object.

Abstract Classes

The root class `Person` does not seem to have any compelling application in its own right. The class serves a useful purpose within the logical specification of an inheritance hierarchy, but instances of it (arguably) do not.

An abstract class is simply a class that exists for high-level organizational purposes, but that cannot ever be instantiated.

In C++, a class is abstract if one or more of its member functions are pure virtual.

A pure virtual member function has no implementation, only a declaration (prototype) in the abstract class declaration.

Pure virtual member functions remain pure virtual in derived classes that do not provide an implementation that overrides the base class prototype.

Using the Abstract Class

Assuming the revised declaration just given, the class `Person` can be derived from, but an attempt to declare an object of type `Person` will generate a compile-time error.

It is, however, legal to declare a pointer to an abstract class, and to use that pointer to store that address of a derived type (as long as it's not also abstract).

Similarly, you can use references to an abstract class, but the target of the reference will always be some derived type.