

Generalization versus Abstraction

Four-Fold Path to Generalization

Hierarchy

Taxonomy

Inheritance

SalariedEmployee Class w/o Inheritance

HourlyEmployee Class w/o Inheritance

What is Common?

What Do We Want?

How Do We Get It?

The Base Class: Employee

A Derived Class: HourlyEmployee

Derived Class Access Privileges

Protected Access

Logical View of an HourlyEmployee  
Object

Using Objects of Derived Classes

Using Objects of Derived Classes

Order of Constructor Execution

A Sibling Class

Assigning Derived Type to Base Type

Assigning Base Type to Derived Type

Parameter Passing Issues

**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” 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” relationship

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

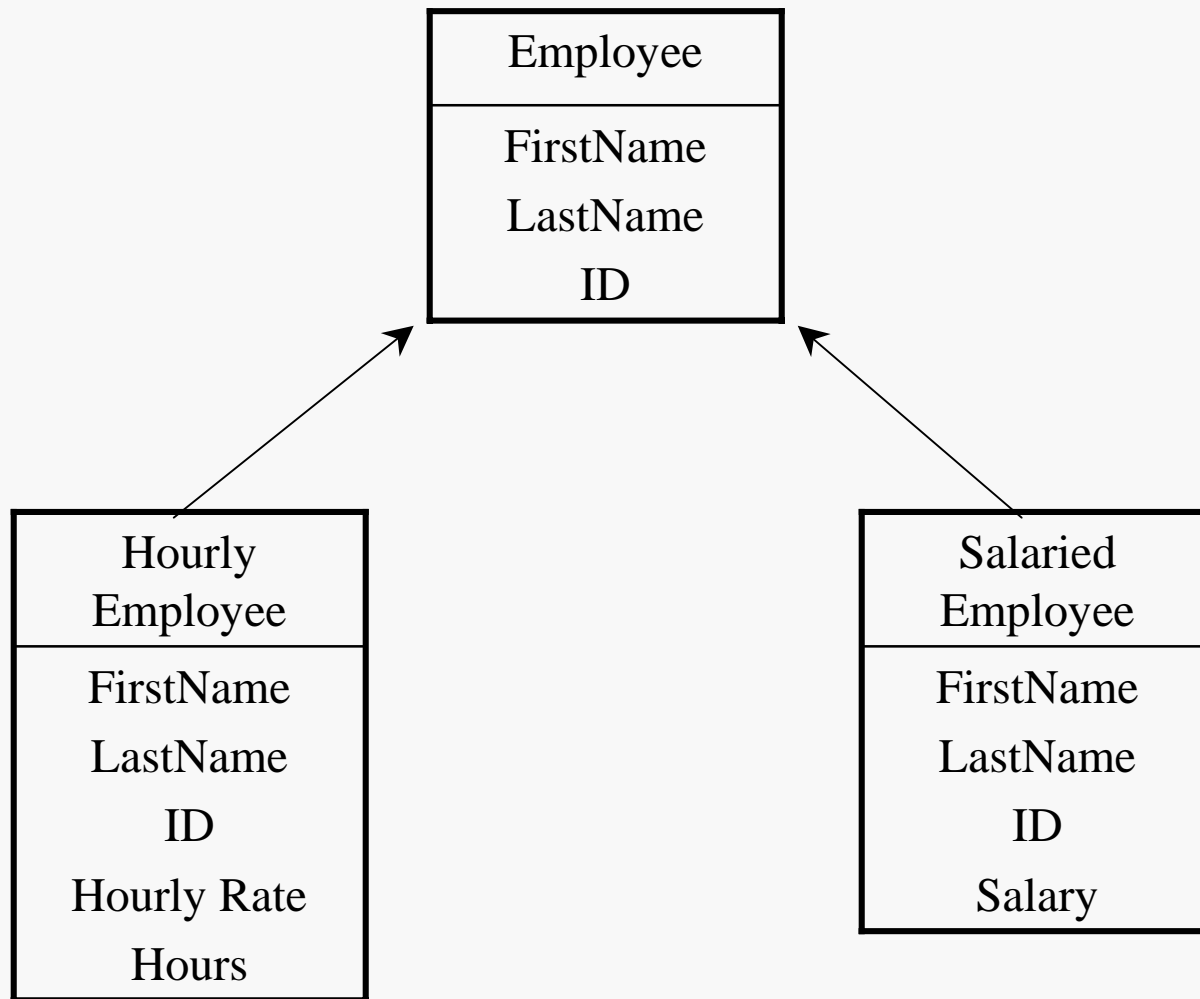
Is a form of knowledge representation – a “taxonomy” structures knowledge about nearby entities.

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” relationships:



## Terminology

- Base type or class (a.k.a. superclass)
- Derived type or class (a.k.a. subclass)

## 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 SalariedEmployee {
private:
    string FName;
    string LName;
    string ID;
    double Salary;
public:
    SalariedEmployee();
    SalariedEmployee(string FN, string LN,
                    string Ident, double S);

    string getName() const;
    string getID() const;
    double getSalary() const;
    void setName(string FN, string LN);
    void setID(string Ident);
    void setSalary(double Sal);
    ~SalariedEmployee();
};
```

```
class HourlyEmployee {
private:
    string FName;
    string LName;
    string ID;
    double Rate;
    double Hours;

public:
    HourlyEmployee();
    HourlyEmployee(string FN, string LN, string ID,
                   double R, double H);

    string getName() const;
    string getID() const;
    void setName(string FN, string LN);
    void setID(string Ident);
    double getRate() const;
    double getHours() const;
    void setRate(double R);
    void setHours(double H);
    ~HourlyEmployee();
};
```

← Specify all the data members

← Specify appropriate constructors

← Specify accessors and mutators for all data members

Both classes contain the data members

```
string FName;  
string LName;  
string ID;
```

and the associated member functions

```
string getName() const;  
string getID() const;  
void setName(string FN, string LN);  
void setID(string Ident);
```

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 identical code.

Simply put, we want to exploit the fact that `SalariedEmployee` and `HourlyEmployee` both are `Employees`.

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

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.

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 either public or private. At this time we will consider only public inheritance.

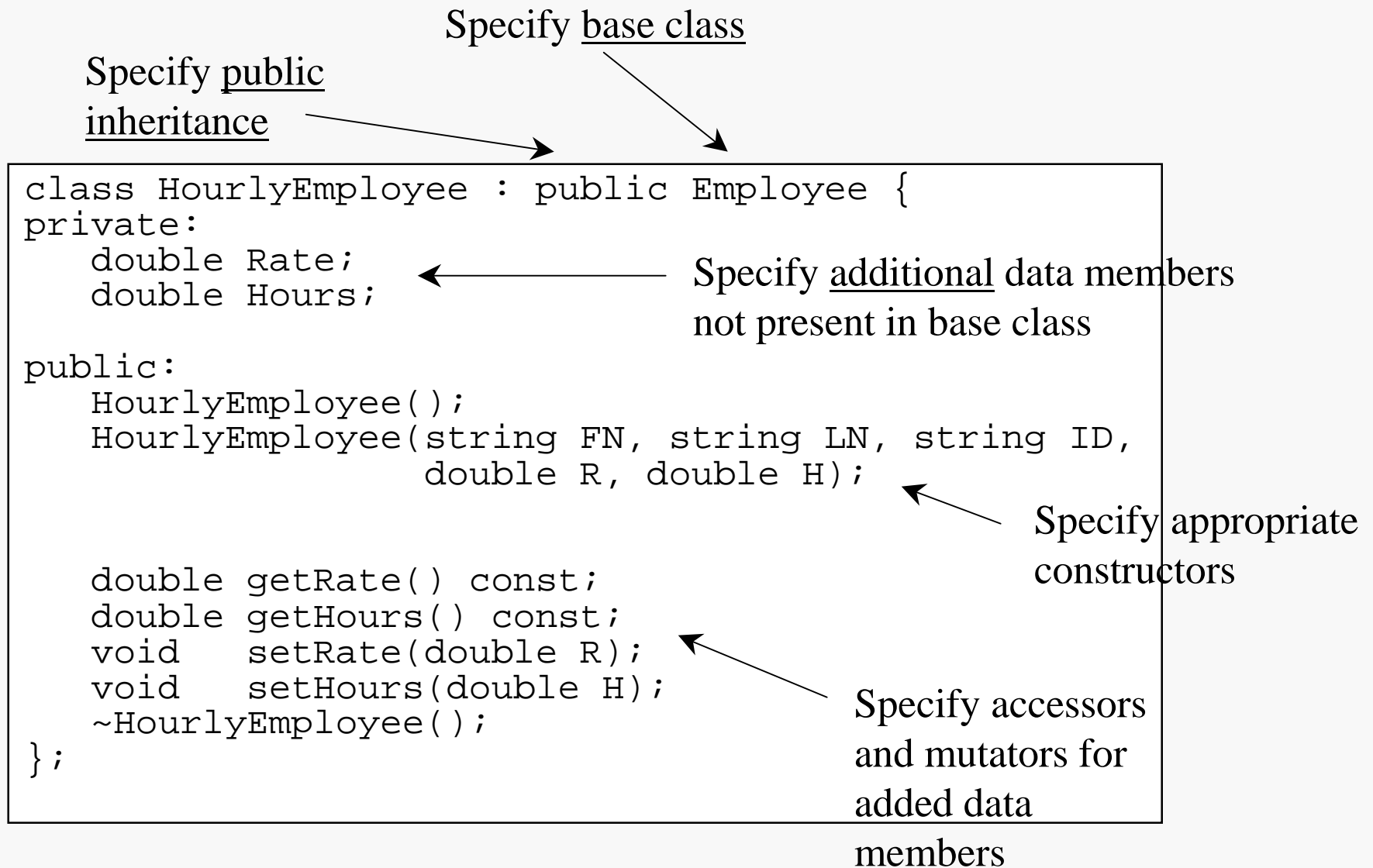
It is also possible for a class to be derived from more than one (unrelated) base class. Such multiple inheritance will be discussed later...

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

```
class Employee {
private:
    string FName;
    string LName;
    string ID;

public:
    Employee();
    Employee(string FN, string LN, string ID);
    string getName() const;
    string getID() const;
    void setName(string FN, string LN);
    void setID(string Ident);
    ~Employee();
};
```

# A Derived Class: HourlyEmployee



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:

```
HourlyEmployee::HourlyEmployee() {  
    FName = "Anonymous";  
    LName = "Person";  
    ID     = "000-00-0000";  
    Rate   = 0.0;  
    Hours  = 0.0;  
}
```

Error: cannot access private member declared in class 'Employee'

This would be allowed:

```
HourlyEmployee::HourlyEmployee() {  
    setName("Anonymous", "Person");  
    setID("000-00-0000");  
    Rate   = 0.0;  
    Hours  = 0.0;  
}
```

However, we'll shortly see a better (and more common) way of doing this.

The fact that derived types cannot access the private members of their base types seems to pose a dilemma. On the one hand, using only public members is unacceptable. On the other hand, the approach used in the corrected constructor on the previous slide is clumsy, at best.

C++ provides a middle-ground level of access control that allows derived types access but denies access by unrelated types.

This is specified using the keyword `protected` to specify the access restrictions for a class member:

```
class Employee {  
protected:  
    string FName;  
    string LName;  
    string ID;  
public:  
    . . .  
};
```



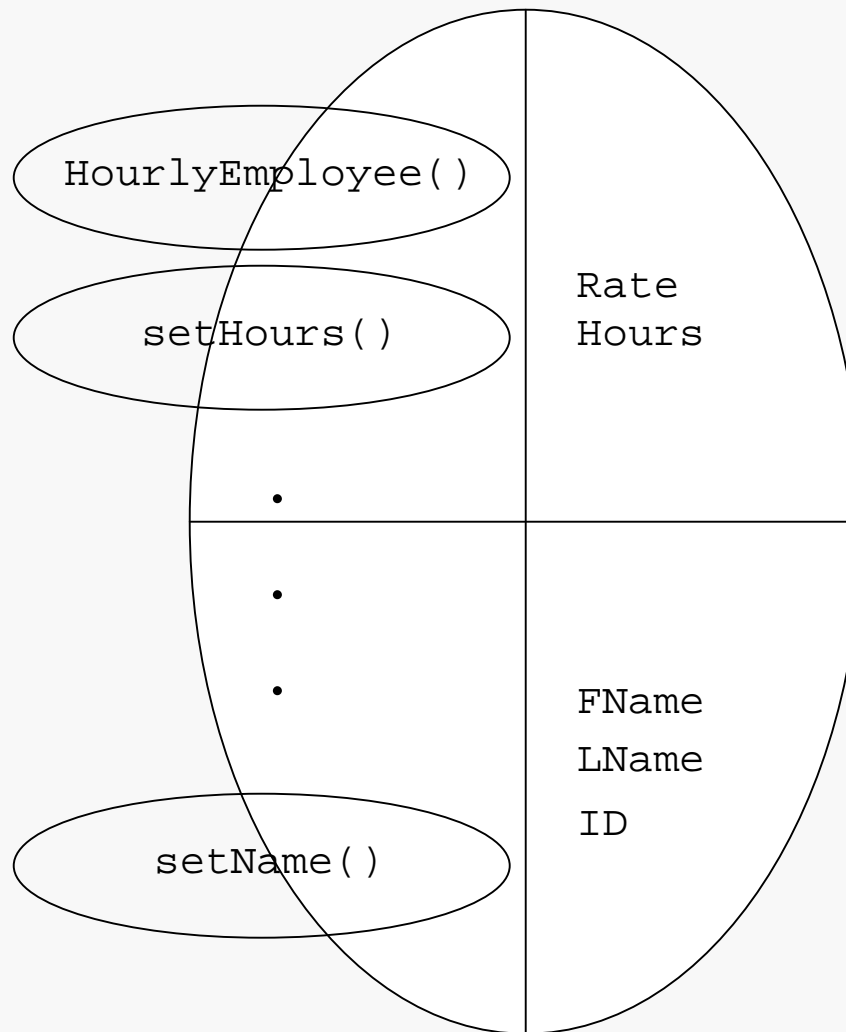
```
HourlyEmployee::HourlyEmployee() {  
    FName = "Anonymous"; // OK now  
    LName = "Person";  
    ID    = "000-00-0000";  
    Rate  = 0.0;  
    Hours = 0.0;  
}
```

HourlyEmployee  
“layer”

Employee “layer”  
inherited from base  
type

Public interface

Private members



```
#include "Employee.h"
#include "HourlyEmployee.h"

void PrintEmployee(Employee toPrint, ostream& Out);
void PrintHourlyEmployee(HourlyEmployee toPrint, ostream& Out);

void main() {

    Employee Me("William", "McQuain", "999-99-9999");
    PrintEmployee(Me, cout);

    HourlyEmployee Homer("Homer", "Simpson", "000-00-0001",
                        13.42, 7.5);

    PrintHourlyEmployee(Homer, cout);

    PrintEmployee(Homer, cout);

}
```

Create and  
print a base  
object...

and a derived  
object...

This is legal. Homer is an instance of HourlyEmployee which is derived from Employee, so Homer IS-AN Employee. However, when passed, Homer is converted (sliced) and the local copy loses the additional members provided by HourlyEmployee.

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

```
void PrintHourlyEmployee(HourlyEmployee toPrint, ostream& Out) {  
    Out.setf(ios::floatfield, ios::fixed);  
    Out.setf(ios::showpoint);  
  
    Out << toPrint.getID();  
    Out << '\t';  
    string Name = toPrint.getName();  
    Out << Name;  
    Out << setw(30 - Name.length())  
        << setprecision(2) << toPrint.getRate();  
    Out << setw(10) << setprecision(2) << toPrint.getHours();  
    Out << '\n';  
}
```

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...).

```
HourlyEmployee::HourlyEmployee() {  
    FName = "Anonymous";  
    LName = "Person";  
    ID     = "000-00-0000";  
    Rate  = 0.0;  
    Hours = 0.0;  
}
```

Redundant: these members would be assigned the same values by the default Employee constructor anyway.

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

```
HourlyEmployee::HourlyEmployee()  
    : Employee("Anonymous", "Hourly", "777-77-7777") {  
    FName = "Anonymous";  
    LName = "Person";  
    ID     = "000-00-0000";  
    Rate  = 0.0;  
    Hours = 0.0;  
}
```

The derived type constructor may specify parameters for the base type constructor, including parameters passed through the derived type constructor:

```
HourlyEmployee::HourlyEmployee(string FN, string LN, string ID,
                                double R, double H)
    : Employee(FN, LN, ID) {

    Rate = R;
    Hours = H;
}
```

The derived type constructor may also invoke constructors for any aggregated data members:

```
HourlyEmployee::HourlyEmployee(string FN, string LN, string ID,
                                double R, double H)
    : Employee(FN, LN, ID), Rate(R), Hours(H) {

    // nothing left to initialize
}
```

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...).

```
HourlyEmployee::HourlyEmployee() {  
  
    FName = "Anonymous";  
    LName = "Person";  
    ID     = "000-00-0000";  
    Rate   = 0.0;  
    Hours  = 0.0;  
}
```

Redundant: these members would be assigned the same values by the default Employee constructor anyway.

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

```
HourlyEmployee::HourlyEmployee()  
    : Employee("Anonymous", "Hourly", "777-77-7777") {  
  
    FName = "Anonymous";  
    LName = "Person";  
    ID     = "000-00-0000";  
    Rate   = 0.0;  
    Hours  = 0.0;  
}
```

```
class SalariedEmployee : public Employee {
private:
    double Salary;
public:
    SalariedEmployee();
    SalariedEmployee(string FN, string LN, string ID, double S);
    double getSalary() const;
    void    setSalary(double R);
    ~SalariedEmployee();
};
```

SalariedEmployee and HourlyEmployee are both derived from the base type Employee (as forecast on slide 5).

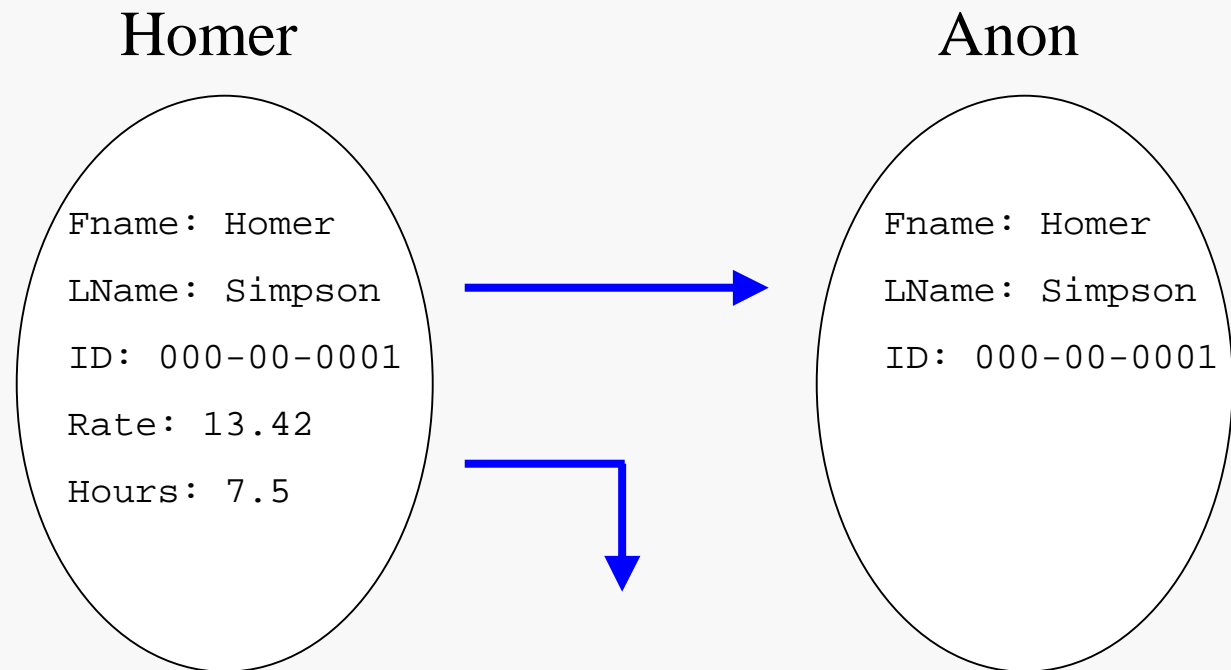
Note that, so far as the language is concerned, SalariedEmployee and HourlyEmployee enjoy no special relationship as a result of this.

By default, a derived type object may be assigned to a base type object:

```
HourlyEmployee Homer("Homer", "Simpson", "000-00-0001", 13.42,  
                    7.5);  
Employee Anon;  
Anon = Homer;  
PrintEmployee(Anon, cout);
```

However...

the base type object will receive only the appropriate "slice" of the derived type object.

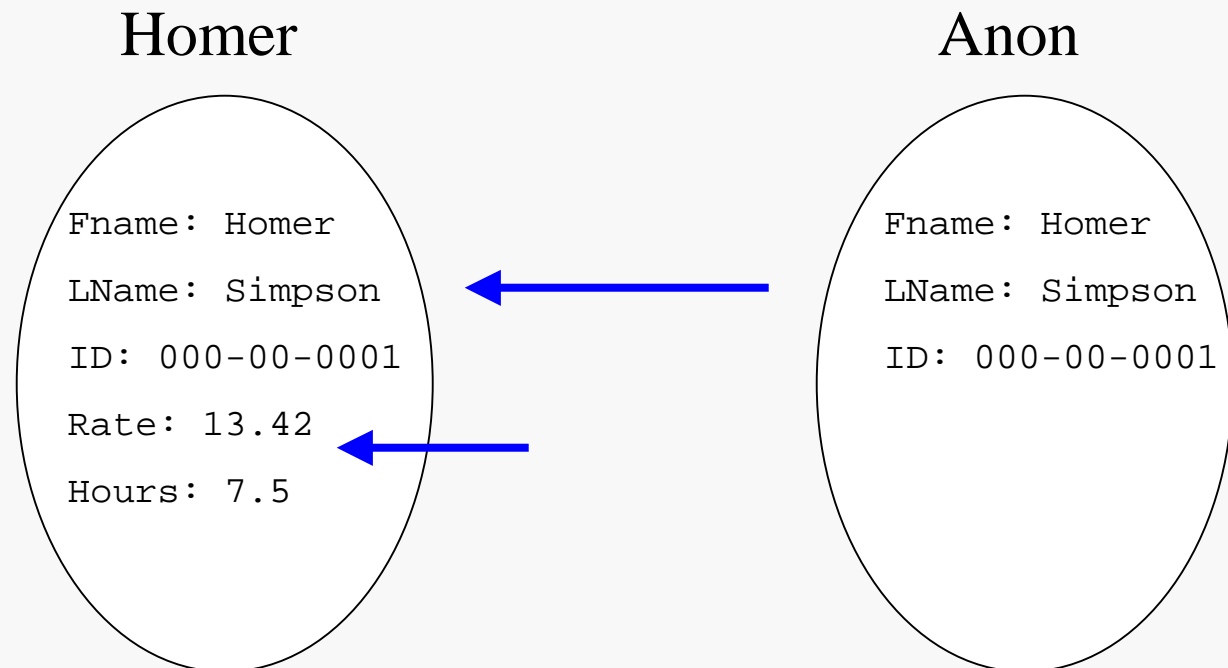


By default, a base type object may not be assigned to a derived type object:

```
Employee Homer("Homer", "Simpson", "000-00-0001");  
HourlyEmployee Anon;  
  
Anon = Homer;    // 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 when passing an object as a parameter.

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

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

```
Employee      Homer("Homer", "Simpson", "000-00-0001");
HourlyEmployee Fred("Fred", "Flintstone", "000-00-0002",
                   17.50, 42.5);

PrintEmployee(Fred, cout);           // legal
PrintHourlyEmployee(Homer, cout);    // illegal
```