WORKED EXAMPLE 10.1

Implementing an Employee Hierarchy for Payroll Processing

Jose Luis Pelaez Inc./Getty Images, Inc.

Problem Statement Your task is to implement payroll processing for different kinds of employees.

- *Hourly employees get paid an hourly rate, but if they work more than 40 hours per week, the excess is paid at "time and a half".*
- *Salaried employees get paid their salary, no matter how many hours they work.*
- *Managers are salaried employees who get paid a salary and a bonus.*

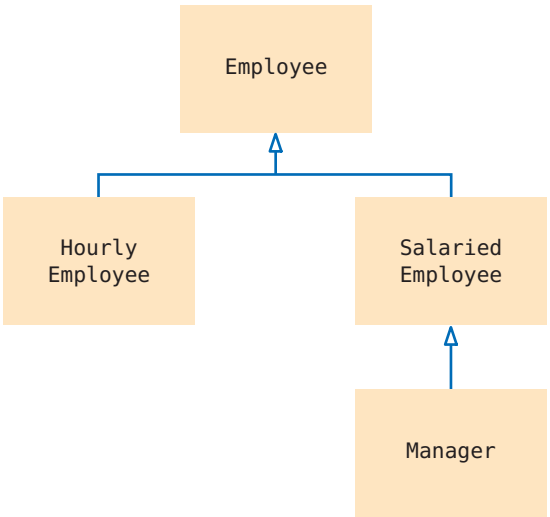
Your program should compute the pay for a collection of employees. For each employee, ask for the number of hours worked in a given week, then display the wages earned.

Step 1 List the classes that are part of the hierarchy.

In our case, the problem description lists three classes: `HourlyEmployee`, `SalariedEmployee`, and `Manager`. We need a class that expresses the commonality among them: `Employee`.

Step 2 Organize the classes into an inheritance hierarchy.

Here is the UML diagram for our classes.



Step 3 Determine the common responsibilities of the classes.

In order to discover the common responsibilities, write pseudocode for processing the objects.

```
For each employee
   Print the name of the employee.
   Read the number of hours worked.
   Compute the wages due for those hours.
```

We conclude that the `Employee` base class has these responsibilities:

```
Get the name.
Compute the wages due for a given number of hours.
```

Step 4 Decide which functions are overridden in derived classes.

In our example, there is no variation in getting the employee's name, but the salary is computed differently in each derived class. Therefore, we will declare the `weekly_pay` member function as `virtual` in the `Employee` class.

```
class Employee
{
public:
   Employee();
   string get_name() const;
   virtual double weekly_pay(int hours_worked) const;
   . . .
private:
   . . .
};
```

Step 5 Define the public interface of each class.

We will construct employees by supplying their name and salary information.

```
HourlyEmployee(string name, double wage);
SalariedEmployee(string name, double salary);
Manager(string name, double salary, double bonus);
```

These constructors need to set the name of the `Employee` base object. We will supply an `Employee` member function `set_name` for this purpose. In this simple example, no further member functions are required.

Step 6 Identify data members.

List the data members for each class. If you find a data member that is common to all classes, be sure to place it in the base of the hierarchy.

All employees have a name. Therefore, the `Employee` class should have a data member `name`. (See Figure 7.) What about the salaries? Hourly employees have an hourly wage, whereas salaried employees have an annual salary. While it would be possible to store these values in a data member of the base class, it would not be a good idea. The resulting code, which would need to make sense of what that number means, would be complex and error-prone.

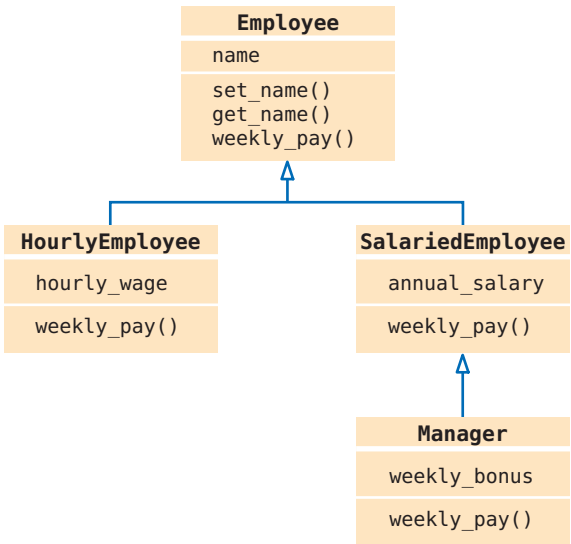


Figure 7 Employee Payroll Hierarchy

Step 7 Implement constructors and member functions.

In a derived-class constructor, we need to remember to set the data members of the base class.

```
SalariedEmployee::SalariedEmployee(string name, double salary)
{
   set_name(name);
   annual_salary = salary;
}
```

Here we use a member function. Special Topic 10.1 shows how to invoke a base-class constructor. We use that technique in the `Manager` constructor:

```
Manager::Manager(string name, double salary, double bonus)
   : SalariedEmployee(name, salary)
{
   weekly_bonus = bonus;
}
```

The weekly pay needs to be computed as specified in the problem description:

```
double HourlyEmployee::weekly_pay(int hours_worked) const
{
   double pay = hours_worked * hourly_wage;
   if (hours_worked > 40)
   {
      pay = pay + ((hours_worked - 40) * 0.5) * hourly_wage;
   }
   return pay;
}

double SalariedEmployee::weekly_pay(int hours_worked) const
{
   const int WEEKS_PER_YEAR = 52;
   return annual_salary / WEEKS_PER_YEAR;
}
```

In the case of the `Manager`, we need to call the version of `weekly_pay` from the `SalariedEmployee` base class:

```
double Manager::weekly_pay(int hours) const
{
   return SalariedEmployee::weekly_pay(hours) + weekly_bonus;
}
```

Step 8 Allocate objects on the free store and process them.

In our sample program, we populate a vector of pointers and compute the weekly salaries:

```
vector<Employee*> staff;
staff.push_back(new HourlyEmployee("Morgan, Harry", 30));
. . .
for (int i = 0; i < staff.size(); i++)
{
   cout << "Hours worked by " << staff[i]->get_name() << ": ";
   int hours;
   cin >> hours;
   cout << "Salary: " << staff[i]->weekly_pay(hours) << endl;
}
```

worked_example_1/salaries.cpp

```
1  #include <iomanip>
2  #include <iostream>
3  #include <vector>
```

```
 4
 5  using namespace std;
 6
 7  /**
 8     An employee with a name and a mechanism for computing weekly pay.
 9  */
10  class Employee
11  {
12  public:
13     /**
14        Constructs an employee with an empty name.
15     */
16     Employee();
17
18     /**
19        @param employee_name the name of this employee
20     */
21     void set_name(string employee_name);
22
23     /**
24        @return the name of this employee
25     */
26     string get_name() const;
27
28     /**
29        Computes the pay for one week of work.
30        @param hours_worked the number of hours worked in the week
31        @return the pay for the given number of hours
32     */
33     virtual double weekly_pay(int hours_worked) const;
34  private:
35     string name;
36  };
37
38  Employee::Employee()
39  {
40  }
41
42  void Employee::set_name(string employee_name)
43  {
44     name = employee_name;
45  }
46
47  string Employee::get_name() const
48  {
49     return name;
50  }
51
52  double Employee::weekly_pay(int hours_worked) const
53  {
54     return 0;
55  }
56
57  //.....................................................................
58
59  class HourlyEmployee : public Employee
60  {
61  public:
62     /**
63        @param name the name of this employee
```

```
64          @param wage the hourly wage
65       */
66       HourlyEmployee(string name, double wage);
67
68       virtual double weekly_pay(int hours_worked) const;
69    private:
70       double hourly_wage;
71    };
72
73    HourlyEmployee::HourlyEmployee(string name, double wage)
74    {
75       set_name(name);
76       hourly_wage = wage;
77    }
78
79    double HourlyEmployee::weekly_pay(int hours_worked) const
80    {
81       double pay = hours_worked * hourly_wage;
82       if (hours_worked > 40)
83       {
84          pay = pay + ((hours_worked - 40) * 0.5) * hourly_wage;
85       }
86       return pay;
87    }
88
89    //.....................................................................
90
91    class SalariedEmployee : public Employee
92    {
93    public:
94       /**
95          @param name the name of this employee
96          @param salary the annual salary
97        */
98       SalariedEmployee(string name, double salary);
99
100      virtual double weekly_pay(int hours_worked) const;
101   private:
102      double annual_salary;
103   };
104
105   SalariedEmployee::SalariedEmployee(string name, double salary)
106   {
107      set_name(name);
108      annual_salary = salary;
109   }
110
111   double SalariedEmployee::weekly_pay(int hours_worked) const
112   {
113      const int WEEKS_PER_YEAR = 52;
114      return annual_salary / WEEKS_PER_YEAR;
115   }
116
117   //.....................................................................
118
119   class Manager : public SalariedEmployee
120   {
121   public:
122      /**
123         @param name the name of this employee
```

```
124         @param salary the annual salary
125         @param bonus the weekly bonus
126      */
127      Manager(string name, double salary, double bonus);
128
129      virtual double weekly_pay(int hours) const;
130   private:
131      double weekly_bonus;
132   };
133
134   Manager::Manager(string name, double salary, double bonus)
135      : SalariedEmployee(name, salary)
136   {
137      weekly_bonus = bonus;
138   }
139
140   double Manager::weekly_pay(int hours) const
141   {
142      return SalariedEmployee::weekly_pay(hours) + weekly_bonus;
143   }
144
145   //.....................................................................
146
147   int main()
148   {
149      vector<Employee*> staff;
150      staff.push_back(new HourlyEmployee("Morgan, Harry", 30));
151      staff.push_back(new SalariedEmployee("Lin, Sally", 52000));
152      staff.push_back(new Manager("Smith, Mary", 104000, 50));
153
154      for (int i = 0; i < staff.size(); i++)
155      {
156         cout << "Hours worked by " << staff[i]->get_name() << ": ";
157         int hours;
158         cin >> hours;
159         cout << "Salary: " << staff[i]->weekly_pay(hours) << endl;
160      }
161
162      return 0;
163   }
```