WORKED EXAMPLE 10.1

Implementing an Employee Hierarchy for Payroll Processing

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.

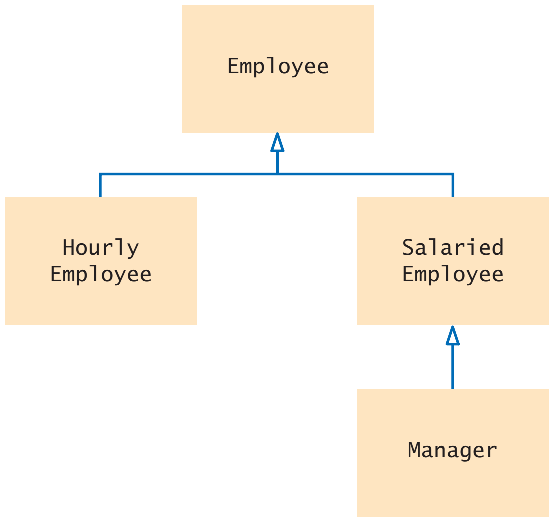
Your program should compute the pay for a collection of employees. For each employee, ask for the number of hours worked in a given week, then display the wages earned.

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.

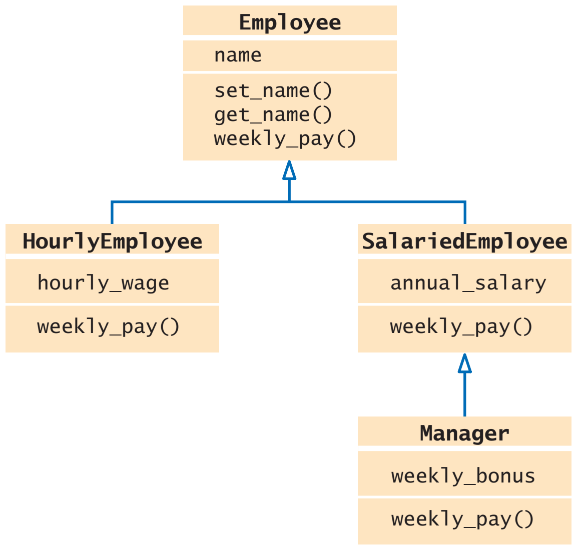


Figure 7 Employee Payroll Hierarchy

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 on page 427 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 heap 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;
}
```

The complete code for this program is contained in `ch10/salaries.cpp`.