Programming Tip 10.2: Don't Use Type Tags

Some programmers build inheritance hierarchies with each object having a `string` tag to indicate its class. They then query that `string`:

```
if (q->get_type() == "Question")
{    // Do something
}
else if (q->get_type() == "ChoiceQuestion")
{    // Do something else
}
```

This is a poor strategy. If a new class is added, then all these queries need to be revised.

In contrast, the addition of a class `NumericQuestion` to our quiz program required no hierarchy rewrite because it uses virtual functions, not type tags.

Don't add type tags to a hierarchy of classes -- use virtual functions instead.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Common Error 10.4: Slicing an Object Function Argument

It is legal to copy a derived-class object into a base-class variable. However, any derived-class information is lost in the process.

To avoid slicing, you can use pointers, as explained with the Quiz array.

Slicing also occurs when a function has a polymorphic parameter (that can belong to a base or a derived class).

Copyright © 2018 by John Wiley & Sons. All rights reserved

References Avoid Slicing an Object Function Argument

An example of a poorly designed function that may slice away required data:

```
void ask(Question q) // Error
{
   q.display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << q.check_answer(response) << endl;
}
```

If you call this function with a `ChoiceQuestion` object, then the parameter variable q is initialized with a copy of that object. But q is a `Question` object; the derived-class information is sliced away.

Instead, use a reference parameter for the object:

```
void ask(const Question& q)
```

Now only the address is passed to the function, no data is sliced away, and the virtual function `display()` works correctly.

Copyright © 2018 by John Wiley & Sons. All rights reserved

Common Error 10.5: Failing to Override a Virtual Function

Two functions can have the same name, provided they differ in their parameter types – this is OVERLOADING, not overriding. For example,

```
class Question
{
public:
   virtual void display() const;
   virtual void display(ostream& out) const;
   . . .
};
```

This differs from **overriding,** where a derived class function re-implements a base class function with the same name and the same parameter types.

Copyright © 2018 by John Wiley & Sons. All rights reserved

Failing to Override a Virtual Function (2)

It is a common error to accidentally provide an overloaded function when you actually mean to override a function. Consider this scary example:

```
class ChoiceQuestion : public Question
{
public:
   void display(); /* Does not override
            Question::display() const */
   . . .
};
```

The display member function in the Question class has subtly different parameters: the implicit parameter `this` is a `const Question*`, whereas in the `ChoiceQuestion` class, the implicit parameter is not `const`.

Copyright © 2018 by John Wiley & Sons. All rights reserved

Override a Virtual Function (3) and the C++11 Fix

In C++ 11, you can use the reserved word **override** to tag any member function that should override a virtual function:

```
class ChoiceQuestion : public Question
{
public:
   void display() override;
   . . .
};
```

If the member function does not override a virtual function, the compiler generates an error. Then the programmer can realize the need to add the missing `const` reserved word.

The compiler also generates an error if you forget to declare the base class member function as `virtual`.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

HOW TO 10.1: Developing an Inheritance Hierarchy (1)

When you work with classes, some of which are general and others more specialized, you want to organize them into an inheritance hierarchy.

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

Step 2: Organize the classes into an inheritance hierarchy.

Step 3: Determine the common responsibilities.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

HOW TO 10.1: Developing an Inheritance Hierarchy (2)

Step 4: Decide which functions are overridden in derived classes.

Step 5: Define the public interface of each derived class

Step 6: Identify data members.

Step 7: Implement constructors and member functions.

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

For a first example of this process, see the textbook for a bank account class hierarchy. The following slides include another example.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (1)

Let's apply the 8-step Inheritance Hierarchy Method to:

Problem Statement: *Implement payroll processing for 3 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.*

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.

Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (2)

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

`HourlyEmployee`

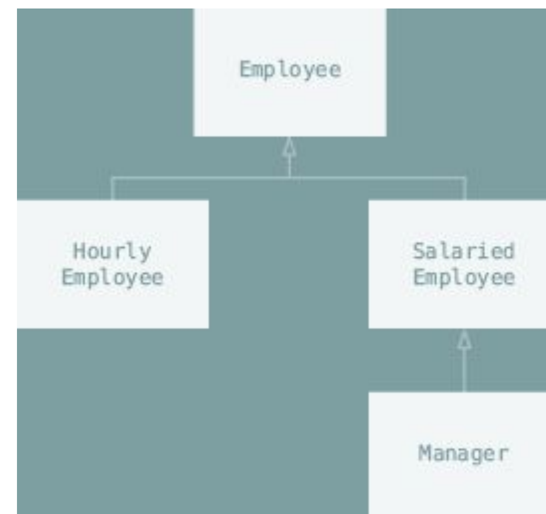
`SalariedEmployee`

`Manager`

AND we need a base class that expresses the commonality among them: **`Employee`.**

2: Organize the classes into an inheritance hierarchy.

Here is the UML diagram:



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (3)

3. Determine the common responsibilities of the classes.

We 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.

Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (4)

4. Decide which functions are overridden in derived classes.

In our example, there is no variation in employee names, but salary is computed differently in each derived class. We will declare the `weekly_pay` member function as `virtual` in `Employee`.

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (5)

5. Define the public interface of each class.

Construct employees from their name and salary:

```
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.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

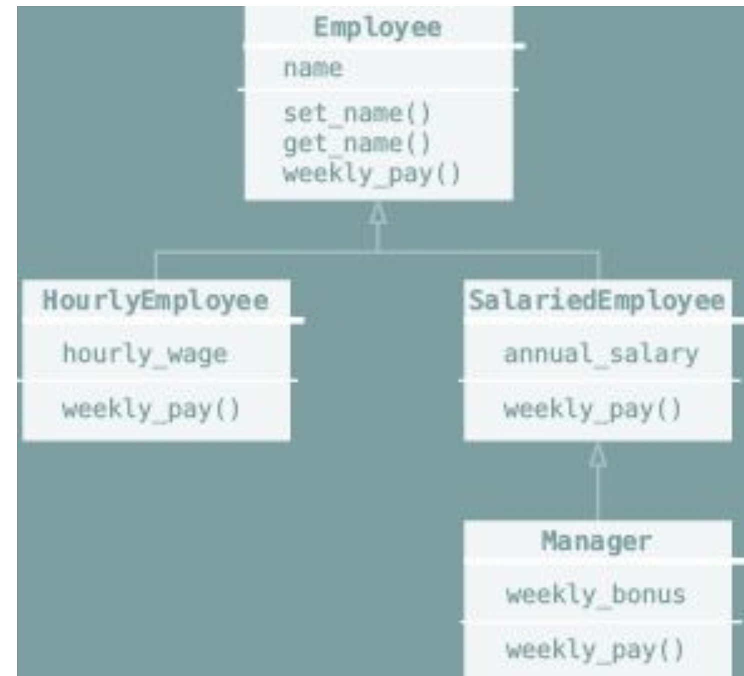
WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (6)

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 class.

All employees have a name. Therefore, the `Employee` class should have a data member `name`.

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.



Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (7a)

7. Implement constructors and member functions.

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

Here we use a member function.

We invoke a base-class constructor for the Manager constructor:

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

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (7b)

7. Implement constructors and member functions.

The weekly pay needs to be computed:

```
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;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (7c)

7. Implement constructors and member functions.

For `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;
}
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

WORKED EXAMPLE 10.1: Employee Hierarchy, Payroll (8)

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;
}
```

The complete code for this program is contained in `worked_example_1/salaries.cpp`.

Copyright © 2018 by John Wiley & Sons. All rights reserved

Chapter Summary (1)

Explain inheritance, base class, and derived class.

- A derived class inherits data and behavior from a base class.
- You can always use a derived-class object in place of a base-class object.

Implement derived classes in C++.

- A derived class can override a base-class function by providing a new implementation.
- The derived class inherits all data members and all functions that it does not override.
- Unless specified otherwise, the base-class data members are initialized with the default constructor.
- The constructor of a derived class can supply arguments to a base-class constructor.

```
Manager::Manager(string name, double salary, double bonus)
   : SalariedEmployee(name, salary)
```

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved

Chapter Summary (2)

Describe how a derived class can override functions from a base class.

- A derived class can inherit a function from the base class, or it can override it by providing another implementation.
- Use `BaseClass::function` notation to explicitly call a base-class function.

Describe virtual functions and polymorphism.

- When converting a derived-class object to a base class, the derived-class data is sliced away.
- A derived-class pointer can be converted to a base-class pointer.
- When a virtual function is called, the version belonging to the actual type of the implicit parameter is invoked.
- Polymorphism (literally, “having multiple shapes”) describes objects that share a set of tasks and execute them in different ways.

Big C++ by Cay Horstmann
Copyright © 2018 by John Wiley & Sons. All rights reserved