UNBC Laboratory Assignment
Computer Science 101—Winter 2007

Virtual Functions

Purpose:

To gain familiarity with the use of virtual functions and programming using inheritance. This laboratory assignment is substantially similar to ones given in previous years.

Due Date:

This assignment is due Wednesday 4 April.

Assignment:

Problem statement

The problem is to implement a payroll processing system that consists of an `Employee` class, and three derived classes: `FullTime`, `PartTime`, and `Contract`. Your task is to write a program that reads information from an `istream` (details about the format of the input information follow) and writes information to an `ostream`.
The output should look like the following:

```
Employee  Employee  Hours/    Rate/   Commision   Other  Deductions  Take
Class     Number    Week      Hour                                   Home
-------------------------------------------------------------------------
FT        00613      40.0     65.00       0.00     0.00      750.00  1850.00
FT        00614      40.0     35.00       0.00     0.00      150.00  1250.00
PT        09312      20.0     15.00       0.00     0.00       15.00   285.00
CO        99001       0.0      0.00       0.00  4512.35        0.00  4512.35
-------------------------------------------------------------------------
Totals:       4     100.0                 0.00  4512.35      915.00  7897.35
```

In order to make the output nicely aligned you should use the `setw` and `setprecision` functions or their equivalents. See Chapter 11 of *Deitel and Deitel* for the details.

Give your `Employee` base class virtual functions to compute hours per week, rate/hour, commission, other, deductions, and take home pay; and a pure virtual[1] function to read employee data from `cin`. In each case, think about how you want to implement these so that the derived classes have the minimum amount to define.

[1] If pure virtual functions haven't been covered in class by the time you start coding this, use an ordinary virtual function. Changing the function to pure virtual later is trivial.

Your "`main`" program *must* use an `Employee*` variable to keep track of the current input employee. After output is displayed for a particular employee, the employee should be deleted; and the next employee record allocated. You probably want your main program to look something like

```
while (cin && !cin.eof())
    {
    EmployeeCode code ;
    Employee* current = 0 ;
    cin >> code ;
    current = get_instance(code) ;
    current->read_details(cin) ;
    cout << *current << endl ;
    delete current ;
    }
```

where `get_instance` is a function that returns an `Employee*` pointer to a newly allocated object of the appropriate derived class.

Use an explicit class called `EmployeeCode` that contains the two-letter codes identifying various categories of employees.

Details for the derived classes are as follows:

FullTime Each full time employee works 38 hours per week. Employees in this class never earn commissions or other payments. Deductions are calculated as $10.00 per week for each dollar per hour earned between $20.00/hr and $35.00/hr; and $20.00 per week for each dollar per hour over thirty-five. Thus someone earning $65.00/hr pays $15 \times \$10.00 + 30 \times \$20.00 = \$750.00$ per week in deductions.

A typical input record for a full time employee looks like:

```
FT 00613 65.00
```

PartTime Each part time employee works between 10 and 30 hours per week. Employees in this class never earn commissions or other payments. Deductions are calculated as 15% of total earnings in excess of $200 per week but less than $500 a week, and 25% on everything in excess of $500 per week. A typical input record for a part time employee looks like:

```
PT 00613 20 15.00
```

Contract Contract employees are never paid an hourly rate and never earn commissions. Furthermore, no deductions are made from their pay. A typical input record for a contract employee looks like:

```
CO 99001 4512.35
```

Implementation

⇒ Implement and test the above program.

You may assume that input comes from `cin` and that output goes to `cout` in your main program if you so choose, but the rest of the functions that read or write should be passed `istream` or `ostream` arguments.

Your main program *must* use an `Employee*` pointer, and the actual employee-derived objects must be created on the heap.

Do *not* put protected or private member variables for each of the `Employee`'s virtual attributes in the `Employee` class. Instead include these variables as private member variables in the appropriate derived class.

Do use virtual member functions for each of the attributes of an `Employee`, in particular, those that show up on the print-out.

Give each class a separate `.h` and `.cpp`-file. If possible, use a `Makefile` for your program. When you have finished designing and testing your program prepare a script file that shows your code and some sample input and output. Be sure to document your code properly.

The following part of the assignment is optional. There are some important ideas and programming techniques here, but you may find that you don't have time to complete this by the end of the course.

The goal of the next part of this assignment is to re-write the program above so that in order to add a new class derived from `Employee` it is sufficient to compile a `.o`-file for the new class and link it with the pre-existing `.o`-files. In particular, you shouldn't have to modify any pre-existing code or recompile any pre-existing `.o`-files.

If you have done the previous parts correctly, you should be close to this goal. In fact, only the `get_instance` function should need modification each time you add a class. If your main program is more tightly linked to the possible classes than this, carefully review how you are using virtual functions and change your program.

In order to change your program so that the `get_instance` function does not require change

each time you add a class, you need to do some work.

"Virtual" constructors

First you need to add an `Employee* clone() const` member function to each class derived from `Employee` that returns a copy of the object that calls it. There is no such thing as a virtual constructor; and this is about as close as you can get. As an example the `FullTime::clone` function might be defined as

```
Employee*
FullTime::clone() const { return new FullTime(*this) ; }
```

Note the upcasting of a `Fulltime*` to an `Employee*`.

Making the `EmployeeCode` class keep track of `Employee` classes

The next trick is to modify the `EmployeeCode` class. The idea here is that the `EmployeeCode` class will have static member variables and static member functions that keep track of the relation between specific `EmployeeCode`'s and the derived class that represents them. By adding a new constructor that also takes an `Employee*` argument, you can associate a *class* with an `EmployeeCode` by storing an `Employee*` pointer to an object of the class in some kind of table that connects `EmployeeCode`'s and `Employee*`'s.

Suppose we add a constructor and static members to an `EmployeeCode` class something like:

```
class Registration ;
class EmployeeCode
{
public:
    EmployeeCode(const char*) ;
    EmployeeCode(const char* code, Employee* example)
      {   // do the standard stuff
          register_code(*this, example) ;
      }
    static Employee* get_instance(const EmployeeCode& code)
      {   return get_ptr()->get(code) ; }
private:
    static void register_code(const EmployeeCode& code, Employee* e)
      { get_ptr()->register_code(code,e) ; }
    static Registration* get_ptr() ;
} ;
```

(The `Registration` class will be explained below.)

Now you can write your `get_instance` function something like

```
Employee* get_instance(const EmployeeCode& code)
{   return EmployeeCode::get_instance(code)->clone() ; }
```

with some appropriate error checking added.

Now if you add a `static` `EmployeeCode` member variable to each class derived from `Employee` you can ensure that each derived class is registered *before the main program begins* with code like:

```
EmployeeCode PartTime::code("PT", new PartTime()) ;
```

Because `PartTime::code` is constructed before the main program begins, the appropriate connection between `"PT"` and the `PartTime` class will be noted by the `EmployeeCode` class before the program begins. Note that the code identifying this connection is contained entirely in the `PartTime.cpp` file, and requires no explicit coding in any other file!

Implementing Registration

Because of the unpredictable order in which static member variables in different `.o` files are initialized, it is necessary to use a static member function rather than a static `Registration*` member variable in `EmployeeCode`.

However, this function can be written as

```
Registration* EmployeeCode::get_ptr()
{   static Registration* ptr = new Registration() ;
    return ptr ;
}
```

An easy way to code the actual Registration class is to make use of a couple of classes in the Standard Template Library (see *Deitel & Deitel* section 20.3.4). Something like the following will work:

```
#include <map>
class Registration
{
public:
    Employee* get_instance(const EmployeeCode& c)
      {
          return theMap[c] ;
      }
    void register_instance(const EmployeeCode& c, Employee* e_ptr)
      {
          theMap[c] = e_ptr ;
      }
private:
    map<EmployeeCode, Employee*> theMap ;
} ;
```

Of course you should add error checking to the above. A `map` object will return a default value if the key specified hasn't been entered. In the case of `Employee*` objects, the default value is a null pointer. In order for the code to work as written, the EmployeeCode needs to have an overloaded "`<`" operator.

⇒ Modify your previous program to include the ideas suggested above. Then modify your program to add a new class of employee, the commission worker.

Commission Commissioned employees work 37.5 hours per week for a varying hourly rate. In addition they are payed a commission on sales, and possibly other expenses related to travel costs. A typical commission employee input record looks like

```
CM 19.31 4000.00 150.00
```

indicating an employee that earns $19.31 per hour, and had commissions totalling $4000.00 and other expenses totalling $150.00.

Deductions for commissioned employees are the same as for part time employees on hourly wages, together with a flat 15% on commissions earned. There are no deductions on the other payments.

⇒ Produce a script file for your modified program.

⇒ *Carefully describe what parts of the existing program you had to* modify *in order to add commission employees. Comment on how you might have been able to reduce the amount of code you had to modify.*