C++ PRIMER PLUS, 5th EDITION
PROGRAMMING EXERCISES
CHAPTER 14

1. The `Wine` class has a `string` class object member (see Chapter 4) that holds the name of a wine and a `Pair` object (as discussed in this chapter) of `valarray<int>` objects (as discussed in this chapter). The first member of each `Pair` object holds the vintage years, and the second ember holds the number of bottles owned for the corresponding particular vintage year. For example, the first `valarray` object of the `Pair` object might hold the years 1988, 1992, 1996, and the second `valarray` object might hold the bottle counts 24, 48, and 144. It may be convenient for `Wine` to have an `int` member that stores the number of years. Also, some `typedef`s might be useful to simplify the coding.

```
typedef std::valarray<int> ArrayInt;
typedef Pair<ArrayInt, ArrayInt> PairArray;
```

Thus the `PairArray` type represents the type `Pair<std::valarray<int>, std::valarray<int> >`. Implement the `Wine` class by using containment. The class hould have a default constructor and at least the following constructors:

```
// initialize label to 1, number of years to y,
// vintage years to yr[], bottles to bot[]
Wine(const char * l, int y, const int yr[], const int bot[]);
// initialize label to l, number of yeras to y,
// create array objects of length y
Wine(const char * l, int y);
```

The `Wine` class should have a method `GetBottles()` that, given a `Wine` object with `y` years, prompts the user to enter the corresponding number of vintage years and bottle counts. A method `Label()` should return a reference to the wine name. A method `sum()` should return the total number of bottles in the second `valarray<int>` object in the `Pair` object.

The program should prompt the user to enter a wine name, the number of elements of the array, and the year and bottle count information for each array element. The program should use this data to construct a `Wine` object and then display the information stored in the object.

1. (continued)
For guidance, here's a sample test program:

```
// pe14-1.cpp -- using Wine class with containment
#include <iostream>
#include "winec.h"

int main (void)
{
    using std::cin;
    using std::cout;
    using std::endl;

    cout << "Enter name of wine:  ";
    char lab[50];
    cin.getline(lab, 50);
    cout << "Enter number of years:  ";
    int yrs;
    cin >> yrs;

    Wine holding(lab, yrs);    // store label, year, give arrays yrs elements
    holding.GetBottles();      // solicit input for year, bottle count
    holding.Show();            // display object contents

    const int YRS = 3;
    int y[YRS] = 1993, 1995, 1998;
    int b[YRS] = 48, 60, 72;
    // create new object, initialize using data in arrays y and b
    Wine more("Gushing Grape Red", YRS, y, b);
    more.Show();
    cout << "Total bottles for " << more.Label()     // use Label() method
         << ":  " << more.sum() << endl;             // use sum() method
    cout << "Bye\n";
    return 0;
}
```

1. (continued)
And here's some sample output:

```
Enter name of wine: Gully Wash
Enter number of years: 4
Enter Gully WAsh data for 4 year(s):
Enter year: 1988
Enter bottles for that year: 42
Enter year: 1994
Enter bottles for that year: 58
Enter year: 1998
Enter bottles for that year: 122
Enter year: 2001
Enter bottles for that year: 144
Wine:  Gully Wash
          Year       Bottles
          1988       42
          1994       58
          1998       122
          2001       144
Wine:  Gusing Grape Red
          Year       Bottles
          1993       48
          1995       60
          1998       72
Total bottles for Gushing Grape Red:  180
Bye
```

2. This exercise is the same as Programming Exercise 1, except that you should use private inheritance instead of containment. Again, a few `typedef`s might prove handy. Also, you might contemplate the meaning of statements such as the following:

```
PairArray::operator=(PairArray(ArrayInt(), ArrayInt()));
cout << (const string &)(*this);
```

The class should work with the same test program as shown in Programming Exercise 1.

3. Define a `QueueTp` template. Test it by creating a queue of pointers-to-`Worker` (as defined in Listing 14.10) and using the queue in a program similar to that in Listing 14.12.

4. A `Person` class holds the first name and the last name of a person. In addition to its constructors, it has a `Show()` method that displays both names. A `Gunslinger` class derives virtually from the `Person` class. It has a `Draw()` member that returns a type `double` value representing a gunslinger's draw time. The class also has an `int` member representing the number of notches on a gunslinger's gun. Finally, it has a `Show()` function taht displays all this information.

A `PokerPlayer` class derives virtually from the `Person` class. It has a `Draw()` member that returns a random number in the range 1 through 52, representing a card value. (Optionally, you could define a `Card` class with a suit and face value members and use a `Card` return value for `Draw()`.) The `PokerPlayer` class uses the `Person` `show()` function. The `BadDude` class derives publically from the `Gunslinger` and `PokerPlayer` classes. It has a `Gdraw()` member that returns a bad dude's draw time and a `Cdraw()` member that returns the next card drawn. It has an appropriate `Show()` function. Define all these classes and methods, along with any other necessary methods (such as methods for setting object values) and test them in a simple program similar to that in Listing 14.12.

5. Here are some class declarations:

```
// emph.h -- header file for abstr_emp class and children

#include <iostream>
#inlcude <string>

class abstr_emp
{
private:
    std::string fname;   // abstr_emp's first name
    std::string lname;   // abstr_emp's last name
    std::string job;
public:
    abstr_emp();
    abstr_emp(const std::string & fn, const std::string & ln,
              const std::string  j);
    virtual void ShowAll() const; // labels and shows all data
    virtual void SetAll(); // prompts user for values
    friend std::ostream & operator<<(std::ostream & os, const abstr_emp & e);
    // just displays first and last name
    virtual ~abstr_emp() = 0; // virtual base class
};

class employee :  public abstr_emp
{
public:
    employee();
    employee(const std::string & fn, const std::string & ln,
             const std::string  j);
    virtual void ShowAll() const;
    virtual void SetAll();
};

class manager:  virtual public abstr_emp
{
private:
    int inchargeof;          // number of abstr_emps managed
protedted:
    int InChargeOf() const { return inchargeof; } //output
    int & InChargeOf(){ return inchargeof; } // input
public:
    manager();
    manager(const std::string & fn, const std::string & ln,
            const std::string & j, int ico = 0);
    manager(const abstr_emp & e, int ico);
    manager(const manager & m);
    virtual void ShowAll() const;
    virtual void SetAll();
};
```

5. (continued)

```
class fink:  virtual public abstr_emp
{
private:
    std::string reportsto;         // to whom fink reports
protected:
    const std::string ReportsTo() const { return reportsto; }
    std::string & ReportsTo(){ return reportsto; }
public:
    fink();
    fink(const std::string & fn, const std::string & ln,
         const std::string & j, const std::string & rpo);
    fink(const abstr_emp & e, const std::string & rpo);
    fink(const fnk & e);
    virtual void ShowAll() const;
    virtual void SetAll();
};

class highfink:  public manager, public fink // management fink
{
public:
    highfink();
    highfink(const std::string & fn, const std::  string & ln,
             const std::string & j, const std::string & rpo,
             int ico);
    highfink(const abstr_emp & e, const std::string & rpo, int ico);
    highfink(const fink & f, int ico);
    highfink(const manager & m, const std::string & rop);
    highfink(const highfink & h);
    virtual void ShowAll() const;
    virtual void SetAll();
};
```

Note that the class heirarchy uses MI with a virtual base class, so keep in mind the special rules for constructor initialization lists for that case. Also note the presence of some protected-access methods. The simplifies the code for some of the `highfink` methods. (Note, for example, that if `highfink::ShowAll()` simply calls `fink::ShowAll()` and `manager::ShowAll()`, it winds up calling `abstr_emp::ShowAll()` twice.) Provide teh class method implementations and test classes in a program. Here is a minimal test program:

5. (continued)

```
// pe14-5.cpp
// useemp1.cpp -- using the abstr_emp classes

#include <iostream>
using namespace std;
#include "emp.h"

int main(void)
{
    employee em("Trip", "Harris", "Thumper");
    cout << em << endl;
    em.ShowAll();

    manager ma("Amorphia", "Spindragon", "Nuancer", 5);
    cout << ma << endl;
    ma.ShowAll();

    fink fi("Matt", "Oggs", "Oiler", "Juno Barr");
    cout << fi << endl;
    fi.ShowAll();
    highfink hf(ma, "Curly Kew"); // recruitment?
    hf.ShowAll();
    cout << "Press a key for next phrase:\n";
    cin.get();
    highfink hf2;
    hf2.SetAll();

    cout << "Using an abstr_emp * pointer:\n";
    abstr_emp * tri[4] = &em, &fi, &hf, &hf2;
    for (int i = 0; i < 4; i++)
        tri[i]->ShowAll();

    return 0;
}
```

Why is no assignment operator define?

Why are `ShowAll()` and `SetAll()` virtual?

Why is `abstr_emp` a virtual base class?

Why does the `highfink` class have no data section?

Why is only one version of the `operator<<()` needed?

What would happen if the end of the program were replaced this code?

```
abstr_emp tri[4] = {em, fi, hf, hf2};
for(int i = 0; i < 4; i++)
    tri[i].ShowAll();
```