C++ PRIMER PLUS, 5th EDITION
PROGRAMMING EXERCISES
CHAPTER 13

1. Start with the following class declaration:

```
// base class
class Cd { // represents a CD stack
private:
    char performers[50];
    char label[20];
    int selections;       // number of selections
    double playtime;      // playing time in minutes
public:
    Cd(char * s1, char * s2, int n, double x);
    Cd(const Cd  d);
    Cd();
    ~Cd();
    void Report() const;   // reports all CD data
    Cd & operator=(const Cd & cd);
};
```

Derive a `Classic` class that adds an array of `char` members that will hold a string idenfitying the primary work on the CD. If the base class requires that any function sbe virtual, modify the base-class declaration to make it so. If a declared method is not needed, remove it from the defintion. Test your product with the following program:

1. (continued)

```
#include <iostream>
using namespace std;
#include "classic.h"      // which will contain #include "cd.h"
void Bravo(const Cd & disk);
{
    Cd c1("Beatles", "Capitol", 14, 35.5);
    Classic c2 = Classic("Piano Sonata in B flat, Fantasia in C", "Alfred Brendel", "phili
    Cd * pcd = &c1;

    cout << "Using object directly:\n";
    c1.Report();          // use Cd method
    c2.Report();          // use Classic method

    cout << "Using type cd * pointer to objects:\n";
    pcd->Report();        // use Cd method for cd object
    pcd = &c2;
    pcd->Report();        // use Classic method for classic object

    cout << "Calling a function with a Cd reference argument:\n";
    Bravo(c1);
    Bravo(c2);

    cout << "Testing assignment:  ";
    Classic copy;
    copy = c2;
    copy.Report();

    return 0;
};

void Bravo(const Cd & disk)
{
    disk.Report();
};
```

2. Do Programming Exercise 1, but use dynamic memory allocation instead of fixed-size arrays for the various strings tracked by the two classes.

3. Revise the `BaseDMA-lacksDMA-hasDMA` class hierarchy so that all three classes are derived from an ABC. Test the result with a program similar to the one in Listing 13.10. That is, it should feature an array of pointers to the ABC and allow the user to make runtime decisions as to what types of objects are created. Add virtual `View()` methods to the class definitions to handle displaying data.

4. The Benevolent Order of Programmers maintains a collection of bottled port. To describe it, the BOP Portmaster has devised a `Port` class, as described here:

```
#include <iostream>
using namespace std;
class Port
{
private:
    char * brand;
    char style[20]; // i.e., tawny, ruby, vintage
    int bottles;
public:
    Port(const char * br = "none", const char * st = "none", int b = 0);
    Port(const Port & p);            // copy constructor
    virtual ~Port() { delete [] brand; }
    Port & operator=(const Port & p);
    Port & operator+=(int b);        // adds b to bottles
    Port & operator-=(int b);        // substracts b from bottles, if available
    int BottleCount() const { return bottles; }
    virtual void Show() const;
    friend ostream & operator<<(ostream  os, const Port & p);
};
```

The `Show()` method presents information in the following format:

```
Brand:  Gallo
Kind:  tawny
Bottles:  20
```

The `operator<<()` function presents information in the following format (with no newline character at the end):

```
Gallo, tawny, 20
```

4. (continued)
The Portmaster completed the method definitions for the Port class and then derived the VintagePort class as follows before being relieved of his position for accidentally routing a bottle of '45 Cockburn to someone preparing an experimental barbecue sauce:

```
class VintagePort :  public Port // style necessarily = "vintage"
{
private:
    char * nickname;                // i.e., "The Noble" or "Old Velvet", etc.
    int year:                       // vintage year
public:
    VintagePort();
    VintagePort(const char * br, int b, const char * nn, int y);
    VintagePort(const VintagePort & vp);
    ~VintagePort() { delete [] nickname; }
    VintagePort & operator=(const VintagePort & vp);
    void Show() const;
    friend ostream & operator<<(ostream & os, const VintagePort & vp);
};
```

You get a job of completing the VintagePort work.

(1) Your first task is to re-create the Port method definitions because the former Portmaster immolated his upon being relieved.
(2) Your second task is to explain why certain methods are redefined and others are not.
(3) Your third task is to explain why operator=() and operator<<() are not virtual.
(4) Your fourth task is to provide definitions for the VintagePort methods.