Chapter 14: OOP and ADTs

Exercises 14.2

1. One can view a derived class as consisting of two components: the portion inherited (data members and function members defined by the base class) and the portion explicitly declared in the derived class definition. Access to each component is governed by the declared accessibility in the class definitions as well as the type of inheritance.

   Public inheritance allows instances of the derived class to access the public and protected members of its base-class component. For example, object.publicBaseFunction() is legal. Moreover, these members retain their visibility for any descendants of the derived class. This form of inheritance provides the most accessibility to the base-class component; only private members of the base class are not directly accessible in the derived class.

   Protected inheritance allows instances of the derived class to access the public and protected members of its base class component only in a protected manner. Thus, object.publicBaseFunction() is not legal, but within the derived class implementation, publicBaseFunction() would be. Also, any descendants of the derived class would also be able to use the inherited members internally. This form of inheritance allows a derived class to use the public and protected members of the base class in an internal manner but not to open them up to the "outside world."

   Private inheritance essentially prevents the derived class from accessing any members of the base class externally or from passing on access to these members to any descendants. It is not commonly used. An example use is when one wants to completely redefine the interface of the base class but wants to use that interface internally.

2. An is-a relationship refers to a relationship between two objects when one is defined (even if only partially) by the other, i.e., by inheritance. For example, a hunting license is a license. The relationship must be defined by innate characteristics or functionally and not by implementation. For example, a stack may be implemented by an array, but it is not correct to say that a stack is an array.

   A has-a relationship refers to containment — when one object contains another. For example, an employee has a name.

   A uses-a relationship refers to interaction between two objects in which one object employs the functionality of the other; they are not related by inheritance or composition.

3. A rectangle has a line.
4. A square is a rectangle.
5. None of the three relationships.
6. A student has a GPA.
7. A student uses the library.

8. A professor is a university employee.

9. An employee is a person.

10. None of the three relationships.

11. A stack is an ADT.

12. A binary tree is a tree.

13. A binary search tree is a binary tree.

14. A vector is a container.

15. A computer is electronic equipment

16. A computer uses an operating system.

17. None of the three relationships.

18. A computer has a CPU.

Exercises 14.5

1.
/* Administrator.h ------------------------------------------------------
 * Header file for a class Administrator derived from SalariedEmployee
 * that adds the attributes unique to administrators.
 * New operations: A constructor and an output operation.
 *-----------------------------------------------------------------------*/
#include <iostream>
#include "SalariedEmployee.h"

#ifndef ADMINISTRATOR
#define ADMINISTRATOR

class Administrator : public SalariedEmployee
{
 public:
 /***** Function Members *****/
 Administrator (long id = 0, string last = "", string first = "", char initial = ' ', int dept = 0,
 double salary = 0, double bonus = 0);
 /---------------------------------------------------------------------*/
 Administrator constructor.

 Preconditions: None.
 Postconditions: Data members myIdNum, myLastName, myFirstName, myMiddleInitial, myDeptCode, and mySalary are initialized by the SalariedEmployee constructor; myBonus is initialized to bonus (default 0).
virtual void display(ostream & out) const;
/*---------------------------------------------------------------------
Output function member.
Precondition: ostream out is open.
Postcondition: A text representation of this Administrator object
has been output to out.
---------------------------------------------------------------------*/

// ... Other administrator operations ...

private:
/***** Data Members *****/
   double myBonus;
};

//-- Definition of constructor
inline double Administrator::Administrator(
   long id, string last, string first, char initial,
   int dept, double salary, double bonus)
: SalariedEmployee (id, last, first, initial, dept, salary),
   myBonus(bonus)
{ }

//-- Definition of Administrator's display()
inline void Administrator::display(ostream & out) const
{
   SalariedEmployee::display(out);   //inherited members
   out << "Bonus: $" << myBonus << endl;  //local members
}
#endif

2.

/* FactoryWorker.h ----------------------------------------------------
Header file for a class FactoryWorker derived from Employee
that adds the attributes unique to factory workers.
New operations: A constructor and an output operation.
---------------------------------------------------------------------*/

#include <iostream>
#include "Employee.h"

#ifndef FACTORY_WORKER
#define FACTORY_WORKER
#define FACTORY_WORKER

class FactoryWorker : public Employee
{
public:
   /***** Function Members *****/
FactoryWorker(long id = 0, string last = "", string first = ", char initial = ', int dept = 0, double unitPay = 0, int numUnits = 0);

FactoryWorker constructor.
Preconditions: None.
Postconditions: Data members myIdNum, myLastName, myFirstName, myMiddleInitial, and myDeptCode, are initialized by the Employee constructor; myPayPerUnit and myNumberOfUnits are initialized to unitPay (default 0) and numUnits (default 0), respectively.

virtual void display(ostream & out) const;
Output function member.
Precondition: ostream out is open.
Postcondition: A text representation of this FactoryWorker object has been output to out.

private:
***** Data Members *****
double myPayPerUnit;
int myNumberOfUnits;
double myWages;
};

//--- Definition of constructor
inline double FactoryWorker::FactoryWorker(
    long id, string last, string first, char initial,
    int dept, double unitPay, int numUnits)
    : Employee (id, last, first, initial, dept),
    myPayPerUnit(unitPay), myNumberOfUnits(numUnits)
{
    myWages = myPayPerUnit * myNumberOfUnits;
}

//--- Definition of FactoryWorker's display()
inline void FactoryWorker::display(ostream & out) const
{
    Employee::display(out); //inherited members
    out << myNumberOfUnits; //local members
    out << " units at $" << myPayPerUnit
    << " gives wages = $" << myWages << endl;
}

#endif
/* Salesperson.h --------------------------------------------------------
 * Header file for a class Salesperson derived from SalariedEmployee
 * that adds the attributes unique to salespersons.
 * New operations:  A constructor and an output operation.
 *-----------------------------------------------------------------------*/
#include <iostream>
#include "SalariedEmployee.h"

#ifndef SALESPERSON
#define SALESPERSON

class Salesperson : public SalariedEmployee
{
public:
  /***** Function Members *****
Salesperson(long id = 0, string last = ",
    string first = "", char initial = ' ', int dept = 0,
    double salary = 0, double commission = 0);

Salesperson constructor.

Preconditions: None.
Postconditions: Data members myIdNum, myLastName, myFirstName,
    myMiddleInitial, myDeptCode, and mySalary are initialized by
the SalariedEmployee constructor; myCommission is initialized to
    commission (default 0).

virtual void display(ostream & out) const;

Output function member.

Precondition: ostream out is open.
Postcondition: A text representation of this Salesperson object
    has been output to out.

// ... Other administrator operations ...

private:
  /***** Data Members *****
  double myCommission;
};

//-- Definition of constructor
inline double Salesperson::Salesperson(
    long id, string last, string first, char initial,
    int dept, double salary, double commission)
    : SalariedEmployee (id, last, first, initial, dept, salary),
      myCommission(commission)
{ }

3.
//--- Definition of Salesperson's display()
inline void Salesperson::display(ostream & out) const
{
    SalariedEmployee::display(out);        //inherited members
    out << "Commission: $"                 //local members
        << myCommission << endl;
}
#endif

4.

/* Document.h --------------------------------------------------------
   Header file for an abstract class Document,
   Abstract operations that must be provided by a derived class:
      read(), display()
   Other operations provided:
      <<, >>, constructor
   ----------------------------------------------------------------------*/
#include <iostream>
#ifndef DOCUMENT
#define DOCUMENT

class Document
{
public:
    /***** Function Members *****/
    virtual void display(ostream & out) const = 0;
    virtual void read(istream & in) = 0;
    //--- Other operations on documents ---
private:
    /***** Data Members *****/
}; // end of abstract-class declaration

// Output operator
inline ostream & operator<<(ostream & out, const Document & doc)
{
    doc.display(out);
    return out;
}

// Input operator
inline istream & operator>>(istream & in, Document & doc)
{ doc.read(in); return in; }
#endif

Then the class License (and others such as Certificate and Report) can be derived from Document. Each must provide definitions of the purely virtual methods read() and display() if the input and output operators are to work correctly. For example:
class License : public Document
{
public:
    /** Function Members *****/
    virtual void display(ostream & out) const;
    // --- Other operations on licenses ---
private:
    /** Data Members *****/
    long myIdNumber;
    string myLastName,
        myFirstName;
    char myMiddleInitial;
    int myAge;
    Date myBirthDay; // where Date is a user-defined class
    // ... and perhaps other attributes ...
}; // end of class declaration

// Definition of display
void License::display(ostream & out) const
{
    out << myIdNum " " myFirstName " "
        myMiddleInitial " ". " myLastName
        " \nAge: " myAge " Birthdate: "
        myBirthDate; // assumes << defined for Date
}

void License::read(istream & in)
{
    in >> myIdNum >> myFirstName >> myMiddleInitial
        >> myLastName >> myAge >> myBirthDate;
    // assumes >> defined for Data
}

5./* LookAheadStack.h ---------------------------------------------------------
   Header file for a class template LookAheadStack derived from Stack
   that adds the attributes unique to look-ahead stacks.
   New operations: A constructor and a revised push operation.
   -----------------------------------------------------------------------*/

#include <iostream>
#include "DStackT.h" // The stack class template version from
                    // Chapter 8 that uses a dynamic array.
#ifndef LOOK_AHEAD_STACK
#define LOOK_AHEAD_STACK

#define LOOK_AHEAD_STACK

template <typename ElementType>
class LookAheadStack : public Stack<ElementType>
{
public:
    /** Function Members *****/
LookAheadStack();

/*================================================================----
Construct a LookAheadStack object.

Preconditions: None.
Postconditions: An empty look-ahead stack has been constructed.
================================================================----*/

void push(ElementType value);

/*================================================================----
Add value at top of look-ahead stack (if there is room).

Preconditions: value is different from top stack element.
Postconditions: value was added at the top of this LookAheadStack provided it is different from top stack element and there is space; otherwise, a duplicate-element message is displayed in the first case and execution allowed to proceed, and in the second case, a stack-full message is displayed and execution is terminated.
================================================================----*/

private:

/*** Data Members /***/

};  // end of class template declaration

//--- Definition of Constructor ---
template <typename ElementType>
inline LookAheadStack<ElementType>::LookAheadStack()
    : Stack<ElementType>()
{
}

//--- Definition of push() ---
template <typename ElementType>
inline void LookAheadStack<ElementType>::push(const ElementType & value)
{
    if (myArray[myTop] != value)
        Stack<ElementType>::push(value);
    else
        cerr << "Item to be added matches top stack element -- not added\n";
}

#endif