COMP 1633 Assignment 4 ("Hockey List", Revisited)

Given:	Monday, March 7, 2022	
Phase 1 Electronic Copy Due:	Monday, March 14, 2022 before 11:59**pm**	***NO LATES***
Phase 1 Hard Copy Due:	Tuesday, March 15, 2022 before 1:00pm	
Phase 2 Electronic Copy Due:	Monday, March 21, 2022 before 11:59pm	***LATES***
Phase 2 Hard Copy Due:	Tuesday, March 22, 2022 before 1:00pm	***ACCEPTED***

Objectives

- To build problem-solving skills.
- To build C++ programming skills.
- To learn about ADTs.
- To learn about classes and objects.
- To gain experience writing client code that instantiates objects and invokes their methods.
- To gain experience implementing a class.
- To gain experience developing a test plan.
- To gain experience writing good documentation.

The Problem

In an object-oriented design, important data structures are encapsulated in classes. The client code instantiates objects and then communicates with these objects via their public interfaces ("methods"). The client code has no access to an object's private implementations. At run time, an object-oriented system is often viewed as an intercommunicating network of objects and client code.

The solution to assignment 2 did not follow an object-oriented design. Instead, heavy use was made of a concrete linked list of "Player" items. Although an effort was made to group team list utility functions (e.g. for insertion, removal, etc.) in a common "`.cpp`" file module, this list data type was still fully exposed. In other words, the main module (the "client" of the `teamsList` module) had full access to the internal details of the linked list and could have replaced utility functions or added code to work on the list directly. There was no attempt at information hiding or abstraction.

In this assignment, you will rework the Hockey Manager program to use a more object-oriented design. You will use a `HockeyList` class which encapsulates the list together with all necessary list operations. This class will define an abstract data type (ADT).

Phase 1

The first phase involves modifying the main program (client code) from assignment 2 to be object-oriented. The purpose of this phase is to understand the class interface for the `HockeyList` and use it correctly in a client program.

A solution for assignment 2, called `a4p1.cpp`, is provided for you to modify. In the class directory in the `assignments/a4/p1_start` subdirectory, you will find this file along with the `ioutil` files and the header file and object module for the `HockeyList` library. Create a `makefile` that will compile and link all of these files into an executable program. Of course, until you modify `a4p1.cpp`, it will not work.

You are to modify how the program interacts with the teams list – changing it from working with a list in a procedural fashion to an object-oriented one. **The client program logic is correct and the basic program logic is not to be modified in any way – drastically changing the program logic will result in a grade of ZERO for this phase! The logic will need to be modified in some places due to the operations provided by the to do list class (this will be discussed later).**

When modifying the program, you are to comment out the existing line(s) of code and put the replacement code BELOW the commented-out code. Each modification is to be numbered, with the numbering starting from 1 and numbered consecutively from the top of the program down. For example:

```
// 1
//   displayTeams(head, cout);
     teamsList.display(cout);
```

If the segment being removed contains a number of lines of code, the C++ comment form of /* and */ can be used.

If any code becomes irrelevant, it should be commented out – and still numbered!
All of the commented-out code must be left in the program you submit.

If it is necessary to add code that does not replace existing code this also is required to be numbered.

The comment containing the modification number MUST be placed in column one to make them easy to locate.

The HockeyList class

As previously mentioned, the public interface `HockeyList.h` exists and **cannot** be altered in any way. As is common for classes, the public members are documented in the header file. Notice that the private implementations are unspecified – these details will be provided in Phase 2. ***How*** the list is implemented is totally irrelevant to Phase 1, so do not make any assumptions about implementation details!

As you read this section of the specification, you should view the header file on INS so that as you read the descriptions of these member functions, you can view their corresponding prototypes.

It is extremely important that you understand how the public member functions for the `HockeyList` class are called and what they return. Why? Because translating the existing non-object-oriented solution into an object-oriented solution requires you to replace the references to the concrete lists with one or more list member functions. Thus, you will have to determine how the existing operations can be achieved with the provided public member functions. All actions done in the existing code can be achieved using the supplied public member functions - and in a simple, logical fashion!

The `HockeyList` class provides the following public member functions; these exact function names **must** be used – and remember C++ is case-sensitive:

- `int load (istream &infile)`
  Takes an open istream and loads the `HockeyList` and returns the number of items added to the list.
- `void write (ostream &outfile) const`
  Takes an open ostream and writes the contents of the `HockeyList` to the ostream in the same format as the input file.
- `void display (ostream &outfile) const`
  Displays the `HockeyList` to the given output stream with a header for team with existing player(s) and with the position value for eachplayer, starting at 1, i.e. as shown in the assignment 2 specifications.
- `void lookup (int position, PlayerNode &player, bool &found) const`
  Takes a position that must be > 0. If an item is found at this position in the `HockeyList` then found will be returned with the value of true and the details from the player at this position will be copied to the PlayerNode provided by the caller. If this position does not exist in the `HockeyList`, then found will return a value of false and contents of the PlayerNode will be undefined.
  *This is a significant change from assignment 2. Whereas the assignment 2 search function returned a pointer to an actual player node in the linked list, lookup returns a **copy** of the player node. Because there is a big difference between being a pointer to a node and being a node, you will need to make a significant number of minor changes in the solution. (These changes will be changing pointer dereferencing to fields to normal structure dereferencing to fields.) These changes will be tedious and repetitive, but not difficult.*
- `char getTeam (int position)`
  The function returns the team letter for the player at the specified position. This function assumes that a valid list position is specified. If the position is invalid the return value is undefined.
  It may not be obvious why this method exists! Go take a look at the player node structure in HockeyList.h. You'll see that player nodes no longer contain a field indicating which team the player is on!
- `bool move (int position)`
  Moves the player at the specified position to the end of the Injury Reserved list. This function returns true if the player was successful moved; false otherwise.

- `bool activate` *`(int position, char team)`*
  Activates a player on the Injury Reserve list at the given position to the end of the specified team. The function returns true if the player was successfully activated; false otherwise.
- `bool update` *`(int position, int goals, int assists, int goalsAgainst)`*
  Updates the player's statistics at the given position, as long as the player is active. The function returns true if the player's statistics were successfully activated; false otherwise.

Notice that these are basically the list operations, with some changes, from assignment 2. Some of the return values are different, specifically the `move`, `activate` and `update` methods all return a `Boolean` value. The existing logic of the client program does not need to be changed to use these return values, i.e. they can be ignored. It is possible that a client program could use these return values, but they are mainly provided for the testing of these functions.

Phase 1 does **not** involve implementing the class. Therefore, make no assumption on how the to-do list will eventually be implemented!

In order to build a fully functional client program, `HockeyList.o` file has been provided. It is a compiled version of a corresponding .cpp file for the `.h` file. This object module contains stubbed out versions of each public member function. The values returned are as follows.

- `load` *returns 5.*
- `move`
- `activate`
- `update`

  These three functions will return `true` if the supplied position is even and return `false` if the supplied position is odd.
- `lookup` *the reference variables will be a PlayerNode with `{"huh",'f',1, 1,0}` and `true`, respectively.*
- `getTeam` *returns `'I'` if the supplied position is `0`; otherwise it returns `'C'`.*

The `write` and `display` functions will have no effect if called.

When a new list object is created, it will be guaranteed to be empty. Notice that the `destroy_list` function from assignment 2 is not specified - because it is not required! The exact reason for this will be made clear in phase 2.

Since `HockeyList` isn't implemented yet – all its member functions are **stubs!** - the program will not generate useful results. However, the program can still be compiled and linked. This is still useful, since you will at least know that the client program contains no compilation errors. The return values from the stubbed functions have been selected so that you can successfully run the program.

Coding Style & Documentation

You must follow the COMP 1633 coding standards exactly. Poor programming style will be graded harshly. The main program documentation will consist of ONLY an ID block.

Submission & Grading

1. Create a directory called `a4p1`.
2. Copy your `a4p1.cpp`, `HockeyList.h`, `HockeyList.o`, `makefile`, `ioutil.h` and `ioutil.cpp` into the `a4p1` directory.
3. `cd` into `a4p1`.
4. Begin a LINUX `script` session.
5. Use `ls -al` to display the contents of the directory.
6. Use `make all` to build your program.
7. Use `cat` to display your `a4p1.cpp` file.
8. Terminate the `script` session.
9. `cd` to the parent directory of `a4p1`.
10. Submit your assignment electronically using `submit` (the item to submit will be the `a4p1` directory).
11. Print a hard copy of your typescript file.
12. Submit your printout in your professor's blue assignment box (**#4**) outside the Mathematics and Computing Department.