CO3105/4105/4203/7105 Coursework 2

Released Nov 6, 2019 **Deadline Dec 2, 2019 5:00 pm**

This assignment consists of two tasks. General information about marking and submission follows after the description of the two tasks.

Task 1 (25% of the module mark)

The aim of this task is to assess your knowledge in dynamic memory management, classes (including deep copy and move semantics) and operator overloading.

You will complete C++ code for a "*bidirectional expansible array*" data structure, or "biarray" for short.[1] This data structure stores a collection of integers in a way that allows direct indexing of elements like the standard C/C++ arrays. But unlike those fixed-size arrays, biarrays can be extended on "either side", as described in the following sections.

Supported operations

More specifically, it supports the following operations:

- Insertions and deletions:
  - `void push_back(int v)`: insert an integer as the last element of the biarray
  - `bool pop_back()`: remove the last integer from the biarray
  - `void push_front(int v)`: insert an integer as the first element of the biarray
  - `bool pop_front()`: remove the first integer from the biarray

  For example, suppose at some point the biarray contains the integers $2, 4, 5, 3, 7$, in this order. A `push_back(6)` changes it to $2, 4, 5, 3, 7, 6$, while `push_front(6)` changes it to $6, 2, 4, 5, 3, 7$.

  Note that biarrays do NOT need to support insertion or deletion at other positions.

- Value access and modifications:
  - `bool get(int i, int& v)`: store the value of the i-th element in v, by reference
  - `bool set(int i, int v)`: set the value of the i-th element to v
  - Array indexing: get/set the value of the i-th element using the [ ] operator as in standard arrays

- Output and size information:
  - `string print()`: return as a string the list of elements

[1] Nope, that's not its real name, I made it up.

- `int getSize()`: return the current number of elements stored
- `int getCapacity()`: return the capacity of the internal array; see below.

- Comparisons:

  It supports the `==` and `!=` operators to check whether two biarrays have the exact same contents (same set of integers and in the same order).

- Class construction and assignment:

  Finally, the class should have a default constructor, a value constructor that takes a standard C++ integer array as input, a destructor, a copy constructor, a move constructor, a copy assignment operator and a move assignment operator.

Further details of what exactly each function should do is in the `BiArray.h` and `BiArray.cpp` files.

Resizing the data structure

The biarray must support efficient direct get/set of elements, and push/pop at front/back must be efficient "on average".[2] There are a number of ways to achieve this. Although from an object-oriented perspective you should be free to choose how to implement things internally, since this is an assignment you should follow what is described below.

You should use a dynamically-allocated standard C++ array (i.e. one that comes from 'new') to store the elements internally. In what follows it will be referred to as 'the internal array'. At any one moment, the *capacity* of the internal array may be bigger than the number of integers currently stored; there may be empty space on either side of the region that actually stores the elements, so that the resizing operation described below does not need to happen every time some integers are inserted. We will call those spaces 'headroom' and 'tailroom' respectively.

When first created by the default constructor or value constructor, the internal array has some initial capacity that is bigger than the number of elements, so there is space in headroom and tailroom; see `BiArray.h` for further details.

If a push back (resp. push front) is requested and there is space in the tailroom (resp. headroom), the element is simply placed in the next available space. If however the respective head/tailroom is full and there is no more space, the array should be resized (by 'new'-ing another one) to a size three times the number of current number of elements, before the new element is inserted. The constant 3 is defined in `LO_THRESHOLD`.

Conversely, if after popping (back or front) the capacity of the internal array is larger than five times the number of elements, it should be resized to three times the number of elements. The constant 5 is defined in `HI_THRESHOLD`. However, the reduction in capacity should not bring the capacity below the constant `INITIALCAP`.

Since a resizing involves 'new'-ing a new array and copying the elements across, you should take this opportunity to "re-center" all the elements in the new array. In other words, right after each resizing, the sizes of headroom and tailroom should be the same (and except for very small arrays, should be equal to the number of elements).

For illustration, this should be the contents of the internal array when the numbers $0, 1, 2, 3, 4, 5, 6$ are pushed back, in this order, and then seven pop-fronts are applied successively: (X denotes the spaces in headroom and tailroom):

[2] In algorithms analysis, the technical terms for these are 'constant time' for direct access, and 'amortized constant time' for insertion/deletion.

```
[X X 0 X X]
[X X 0 1 X]
[X X 0 1 2]
[X X X 0 1 2 3 X X]
[X X X 0 1 2 3 4 X]
[X X X 0 1 2 3 4 5]
[X X X X X X 0 1 2 3 4 5 6 X X X X X]
[X X X X X X X 1 2 3 4 5 6 X X X X X]
[X X X X X X X X 2 3 4 5 6 X X X X X]
[X X X X X X X X X 3 4 5 6 X X X X X]
[X X X 4 5 6 X X X]
[X X X X 5 6 X X X]
[X X 6 X X]
[X X X X X]
```

What to implement

You should complete the code as specified in the `BiArray.h` and `BiArray.cpp` files.

You must not change the public interface of the class. You will need to add private data members, and you are allowed to add private member functions should you want to. You could add extra public member functions, although I don't see the need for doing so. The given `main.cpp` is an example that illustrates how the functions are to be used.

You must not use anything from the Standard Template Library (STL).

Task 2 (15% of the module mark)

The aim of this part is to assess your knowledge on inheritance and virtual functions.

You are part of a software team designing a prototype fantasy football game, and you are asked to develop C++ code as follows.

What is a fantasy football game

In a real-world football league, there are a number of teams, and each team has a number of players. A football player is of exactly one of four roles: goalkeeper, defender, midfielder, attacker. Each week there is a series of football matches between teams. Each player in a each team will score some points based on their contributions and their team's performances (see below).

In the fantasy game, the 'player' (as in gamer, not footballer) forms a fantasy team by picking 11 players (footballers), possibly from different real-life teams. In this prototype, we assume the composition of the four types of players is arbitrary, as long as they add up to 11. So while normally a team must have one goalkeeper and may have (for example) 4-4-2 as the number of defenders, midfielders and attackers respectively, any combination is allowed in this prototype game.

Scoring

Each footballer is awarded points in the fantasy game based on their contributions such as scoring goals, conceding goals, making assists and so on in the real-life match. The following

table defines the points awarded.

All players:	
Each goal scored by a striker	+4 points
Each goal scored by a midfielder	+5 points
Each goal scored by a defender or a goalkeeper	+6 points
Each assist by any player	+3 points
Midfielders only:	
No goals conceded by their team	+1 point
Defenders and goalkeepers only:	
Every 2 goals conceded by their team	−1 point
No goals conceded by their team	+4 points
Goalkeepers only:	
Every 3 shots saved	+1 point

For example, a defender who scored one goal, made two assists and conceded no goals will get $6 + 3 + 3 + 4 = 16$ points, while a goalkeeper who saved 7 shots, conceded 9 goals and had no other contributions will get $\lfloor 7/3 \rfloor - \lfloor 9/2 \rfloor = 2 - 4 = -2$ points. (The symbol $\lfloor ... \rfloor$ indicates rounding down to the nearest integer.)

Note that goals conceded is a team statistic: all players of the same (real-life) team concede the same number of goals and 'share the responsibility'. All other statistics are for the individual players (those who actually scored the goal or made the assist). Shots saved is a goalkeeper-only individual statistic.

What to implement

The classes `Team`, `Player`, `Goalkeeper`, `Defender`, `Midfielder`, `Attacker` and `FantasyTeam` are defined for you. `Goalkeeper`, `Defender`, `Midfielder` and `Attacker` are derived classes of `Player`; this `Player` class should be an abstract class. `FantasyTeam` contains pointers to 11 `Player`s as its member variables.

The classes have a number of functions for each team/player, such as adding the number of goals scored, returning the score of a player, etc. You should complete the implementations of the functions in these classes as indicated in the `Football.h` and `Football.cpp` files. The given `main.cpp` is an example that illustrates how the functions are to be used.

In addition, you may need to insert the `virtual` keyword or the pure specifier `= 0` at various places. You must turn `Player` into an abstract class (which means you must make some function pure virtual). You must not make any other changes to the public interface of the classes. You may add any protected/private member variables (in any class) as required. You are also allowed to add private or public member functions, or to add friendships among these classes.

When adding member variables or implementing functions, you should try to place them at the highest possible point in the class hierarchy. This will affect the code inspection part of your marks.

Marking criteria and test suites

In each of the two tasks:

- 30% of the marks are awarded for the correct observed bahaviour of your program, judged solely by the passing (or not) of the test cases given. Each task comes with its test suite. More information on how to use the test suites will be explained in class.

Testing will only be conducted in the departmental linux system. You must make sure your programs can be compiled and run in the departmental linux system with the given makefile and test files.

- 60% of the marks are awarded for the correctness of your implementation. This will be done by manual inspection, possibly helped by test cases not released to you before the due date. The marking reflects both errors that lead to observable incorrect behaviour as well as those that don't (such as memory leaks).
- 10% of the marks are awarded for readability of the code and use of good C++ coding style.

Submission instructions

Use the handin system at https://campus.cs.le.ac.uk/handin/

Submit only the files `BiArray.h`, `BiArray.cpp`, `Football.h` and `Football.cpp`. While you may want to change the `main.cpp` files for your own testing, they are not part of the submission. The test suites and the makefiles also should not be submitted.

Anonymous marking is achieved by having only the userid in the submission process. Please do not write down your name or other identifiable information in your submission.