COMP 2012 Object-Oriented Programming and Data Structures

Assignment 1 Tournament Organizer

A chess tournament. Image Source: chess.com.

Introduction

Suppose you are an organizer for some competition with many participants, for example chess, tennis, e-sports, etc. In a tournament, pariticipants compete with each other so that eventually, the best performing player can be determined as the winner. You would like to design a program that will help you with deciding the match pairing for the participants following a specific set of rules. Therefore, in this assignment, you will create a simple program to pair players in a Swiss-system format, and you will need to incorporate knowledge of dynamic object allocation and deallocation, and class constructors and destructor.

Swiss-system tournament

There are multiple ways a tournament can be conducted in order to find a winning player, resulting in different total number of matches. The most popular systems used nowadays include:

- Knockout system, or single-elimination system, puts participants in brackets and compete with each other. Participants losing any matches will be eliminated from the tournament and cannot play any more games. This system ensures the winner can be unanimously determined using fewest number of games, but will also result in unbalanced number of games for players, as well as the potential of very good players being knocked out due to a single underperforming game. Some tournaments alleviate this problem by allowing "second-chances" via a secondary bracket for losers, called double-elimination tournament.
- Round-robin system pairs every players with each other, ensuring that all possible match-ups are played and all participants play the same number of games. Points are assigned to winners of matches, and are split in drawn games. However, the total number of games can be exceedingly high for a large number of participants.
- Swiss system also pairs players and do not eliminate anyone like round-robin system, but the number of rounds is drastically reduced by taking into account performance of players. In general, players with similar score/performance are paired against each other, and the winner is determined as the best performing player after sufficient number of rounds. The pairing rules for this system can get quite complicated, so we will implement a simple version of the Swiss system in this assignment.

Classes overview

The following section describes the given classes.

Player

```
class Player {
  private:
      char* name;
      int elo;
      int score;

  public:
      // TASK 1: Implement the conversion constructor and destructor
      Player(const char* const name, const int elo);
      ~Player();

      // The following copy constructor and assignment operator are deleted
      Player(const Player& player) = delete;
      Player& operator=(const Player&) = delete;

      // The accessor, mutator and print functions are given
      int getELO() const { return elo; }
      int getScore() const { return score; }

      void addScore(const int points) { score += points; }

      void print() const {
          std::cout << name << " (" << elo << ")";
      }
};
```

The Player class represents a participant in the tournament. `name` is the player's name. `elo` is the player's rating score, which is used as a secondary parameter to rank players for pairing purpose. A higher ELO rating means the player is considered "stronger", but it doesn't mean said player will always win against lower ranked opponents. Lastly, `score` is the player's score in the tournament, which starts at 0. For this assignment, the player gains 2 points for winning a match, and 1 point for a draw.

For Task 1, you will need to implement the conversion constructor and destructor to correctly handle the data members. All the other member functions have been implemented in **player.h**.

Note: You may notice that the copy constructor and assignment operator are declared as `= delete` (you will learn about assignment operator later in the course). This is a C++11 feature allowing us to explicitly state that this class has no copy constructor and assignment operator. We don't want to create copied instances of players to prevent the case of modifying a copy's score but not the original - if there is a player named "Alice", there should be only one such instance of Player with the data member `name` pointing to a string "Alice". Therefore, ***you should not define them*** *in your solution.*

PlayerList

```
class PlayerList {
  private:
      int numPlayers;
      Player** players;

  public:
      // TASK 2: Implement the default constructor, conversion constructor and destructor
      PlayerList();
      PlayerList(const PlayerList& list);
      ~PlayerList();

      // The following accessor functions are given
      int getNumPlayers() const { return numPlayers; }
      Player* getPlayer(const int index) const { return players[index]; }

      // TASK 3: Implement PlayerList insert, sort and splice
      void addPlayer(Player* const player);
      void sort();
      PlayerList* splice(const int startIndex, const int endIndex) const;
};
```

The PlayerList class is a data structure to store a variable number of Players. Because the number of players can be arbitrary, and we will often need to rank players according to their performance, we design the class to be a variable sized array and support 2 mutator functions: insertion via `addPlayer()` and sorting elements via `sort()`. An additional function `splice()` is for retrieving a subset of the array of Players.

For Task 2, you will need to implement the constructors and destructor of this class. `players` is a **1D array of pointers to Player objects** (not a 2D array), and `numPlayers` is the number of players stored in the list. Note that in the copy constructor, you should *assign the pointers to point to the same object as the passed `list`'s pointers*, instead of creating new copies of Player instances (recall that the Player copy constructor and assignment operator are not available). This is because there can only be 1 Player object for each player, but there can be multiple lists of players storing the same Player, and modifying said Player's `score` in one list will also update the other lists. You will still need to dynamically allocate memory for the array of pointers, so that any other modifications to a list (e.g. sorting or adding player) will not affect other lists.

For Task 3, you will need to complete the `addPlayer()`, `sort()` and `splice()` member functions, according to the task description. Other accessors have been provided in **player.h**, allowing other classes to access the Player pointers similar to an array. We ignore the scenario of any player withdrawing in this assignment, hence element deletion method is not needed.

Match

```
class Match {
  private:
      Player* p1;
      Player* p2;

  public:
      Match(Player* const p1, Player* const p2);

      void play();
};
```

The Match class handles a game between 2 Players, including message printing, input handling and score assigning. When a Match is played via `play()`, the user can input a number from 1 to 3 as the outcome of the match: whether the first player wins, second player wins, or the match ends in a draw. In practice, you can think of this as the program waiting for the actual outcome of the assigned match, then using the result to decide the pairings for next round.

This class has been fully implemented in **match.h** and **match.cpp**.

Swiss

```
class Swiss {
  private:
      int curRound;
      int numRounds;
      PlayerList list;

  public:
      // TASK 4: Implement the conversion constructor
      Swiss(const int numRounds, const PlayerList& list);

      // TASK 5: Implement the function to conduct the Swiss tournament
      void play();

      // The leaderboard print function is given
      void printLeaderboard() const {
          std::cout << "==================================" << std::endl;
          std::cout << "ROUND " << curRound << "/" << numRounds << " ENDED." << std::endl;
          for (int i=0; i<list.getNumPlayers(); ++i) {
              list.getPlayer(i)->print();
              std::cout << ": " << list.getPlayer(i)->getScore() << std::endl;
          }
          std::cout << "==================================" << std::endl;
      }
};
```

This class handles a complete tournament in Swiss format. `numRounds` is the total number of rounds, and `curRound` is the current round number, which starts at 0 (since the tournament has not started). `list` is a PlayerList storing all Players. In an actual tournament, the number of rounds is calculated using the number of players, but for our assignment we will let the user input any (positive) number of rounds for the tournament.

*Hint: `list` is a **static-allocated** data member of `Swiss`, so you should not use `new` or `delete` for this data member.*

For Task 4, you will need to implement the conversion constructor to initialize the data members. And for Task 5, you need to write the function `play()` to simulate all rounds of the tournament according to the (simplified) Swiss rules described in the section below. This function will update `curRound` accordingly, and you will need to make use of the provided `printLeaderboard()` function in **swiss.h** to print the list of players and their scores after each round.

Additional remarks

We value academic integrity very highly. Please read the Honor Code section on our course webpage to ensure you understand what is considered plagiarism and the penalties. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, including expulsion.

End of Introduction

print functions using `cout` are defined in the header files. Since you are not required to implement or change any print functions, this

Task description

You will need to complete the following tasks in 2 new files: **player.cpp** and **swiss.cpp**. Complete the relevant tasks in the relevant files. In addition, **you are not allowed to include additional libraries unless specified**. You may only `#include` any given header files (**match.h**, **player.h**, **swiss.h**), or the **<cstring>** library to make use of `strlen()` and `strcpy()` functions. You are also allowed to include **<iostream>** for debug purposes.

Task 1

Implement class Player's conversion constructor and destructor in **player.cpp**. You will need to allocate memory for the string `name`, and you can use `strlen()` and `strcpy()` functions to help with copying data from the parameter to the data member. `score` should be initialized to 0. The destructor should free any memory allocated in the constructor.

Task 2

Implement class PlayerList's default constructor, copy constructor and destructor in **player.cpp**. The default constructor simply sets the number of players to 0 and assign `nullptr` to the `players` array. The copy constructor allocates memory for the `players` array and copy data from the passed list to the new array. Finally, free up this memory in the destructor.

*Note: **you should not delete the Player objects pointed at by the array of pointers**, because other lists may also be pointing at those instances. The Player objects are created, managed and destroyed in the `main()` function.*

Task 3

Implement the following 3 functions in **player.cpp**:

addPlayer()

```
void PlayerList::addPlayer(Player* const player)
```

This function adds a Player pointer to the list. You will need to handle the dynamically allocated array correctly, increase its size by allocating a new array if necessary.

Note that there is no "mandatory" way you can implement this data structure and method: you may insert the new pointer at the beginning, end or anywhere in the middle; the actual array size can also be larger and have "empty spaces" at the end, as long as `numPlayers` is correctly maintained. After sorting, it should be possible to traverse the array from index 0 to `numPlayers-1` to get all players from the highest rank to the lowest.

sort()

```
void PlayerList::sort()
```

This function sorts the array of Player pointers, according to the Players' `score` and `elo`. The players with the highest scores should be ranked first (low index). For players with the same score, players with higher ELO rating are ranked first. You may assume there are no 2 players with the same ELO rating. After completing this function, you can `sort()` a list to rank all players according to their performance, as well as to "seed" players using their ELO ratings at the beginning of the tournament (when the scores of everyone are 0).

Feel free to look up methods to perform array sorting - there is no time requirement for this function (as long as it doesn't run for too long). Because the ELO ratings of everyone are unique, the sorting result should also be unique regardless of your algorithm. Here are some sorting algorithms you may try to implement:

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort

Here is an example of the expected behaviour for the sorting function. Suppose the PlayerList starts with the following 5 players:

Alice	Bob	Carol	David	Ethan
ELO: 1500	ELO: 1000	ELO: 2000	ELO: 1750	ELO: 1200
Score: 0	Score: 0	Score: 0	Score: 0	Score: 0

We call `sort()` on this PlayerList. Since all players have equal score, we sort them by their ELO (highest ELO first):

Carol	David	Alice	Ethan	Bob
ELO: 2000	ELO: 1750	ELO: 1500	ELO: 1200	ELO: 1000
Score: 0	Score: 0	Score: 0	Score: 0	Score: 0

Now the PlayerList is sorted. Suppose after a round of matches, the scores of players are updated as so:

Carol	David	Alice	Ethan	Bob
ELO: 2000	ELO: 1750	ELO: 1500	ELO: 1200	ELO: 1000
Score: 1	Score: 0	Score: 1	Score: 2	Score: 2

We call `sort()` again. Players with the highest scores will be ranked first, and players with the same score will be ranked by their ELO:

Ethan	Bob	Carol	Alice	David
ELO: 1200	ELO: 1000	ELO: 2000	ELO: 1500	ELO: 1750
Score: 2	Score: 2	Score: 1	Score: 1	Score: 0

This PlayerList is now sorted by the players' performance after one round of matches.

*Hint: `list` is an **array of pointers**. Sorting can therefore be done easily using swaps - if you need to swap the position of 2 Players in the list, you can simply swap the value of the 2 corresponding pointers.*

splice()

```
PlayerList* PlayerList::splice(const int startIndex, const int endIndex) const
```

This function returns a new dynamically allocated instance of PlayerList storing pointers to Players pointed at by the current PlayerList, from index `startIndex` to **the index before** `endIndex`. For invalid indices, you can simply return a default PlayerList with no players. You may find this function useful in carrying out the Swiss format algorithm later.

To illustrate the behaviour of this function, suppose a similar data structure "NumberList" is implemented for integers, and we have the following list:

```
        [4 3 8 9 5 1 2 7 6]
//Index: 0 1 2 3 4 5 6 7 8
//Size: 9
```

Examples of `splice()`:

- `list.splice(0, 4)`: Returns a new NumberList: [4 3 8 9]
- `list.splice(1, 4)`: Returns a new NumberList: [3 8 9]
- `list.splice(3, 8)`: Returns a new NumberList: [9 5 1 2 7]
- `list.splice(-1, 5)` (*`startIndex` < 0*) or `list.splice(7, 10)` (*`endIndex` > size*): Returns default PlayerList
- `list.splice(5, 5)` or `list.splice(4, 3)` (*`startIndex` >= `endIndex`*): Returns default PlayerList

Task 4

Implement class Swiss's conversion constructor in **swiss.cpp**. Note that after initializing `list`, you should perform "seeding" for the tournament by calling `list.sort()`. To validate, after creating a Swiss instance, calling `printLeaderboard()` should print all players in decreasing order of ELO rating.

Task 5

Implement the following function in **swiss.cpp**:

```
void Swiss::play()
```

This function simulates a full tournament in Swiss format. You may follow these steps to implement the function:

For each `curRound` from 1 to `numRounds`:

- Create an array of PlayerList, where each list consists of all players with the same score. Since after each round, each player gains at most 2 points, in round `curRound` there will be `2 * curRound - 1` possible scores the players can have.
  - For example, at the start of round 3, two rounds have passed and players can have maximum score of 4 and minimum score of 0. In total, there are 2 * 3 - 1 = 5 possible scores any player may have.
- Loop over the `list` of players and add them to the corresponding PlayerList depending on their scores using `addPlayer()`. Alternatively, you can make use of `splice()` to quickly generate new PlayerLists (you can assume the current `list` is sorted at this point). Lists corresponding to scores with no players are empty (`numPlayers` = 0).
- For each non-empty PlayerList, **starting from the highest score**:
  - Sort the list
  - Find the player at the middle index. For example, if there are 6 or 7 players, the middle index is 3.
  - Create a Match for each pair of players according to this midpoint, e.g.: Index 0 vs. index 3, index 1 vs. index 4, etc.; and call `play()` on those Matches
  - If there is a player remaining (due to odd number of players), this player has the highest index (e.g. index 6 for 7 players). Add this player to the PlayerList 1 point lower, so that he/she will be paired with the next set of players. If this is the last PlayerList (score = 0), **instead give the player 2 points**. We say that this player received a *bye* to the next round.
- Sort the `list` after all matches have been played, then call `printLeaderboard()`.
- Delete any memory dynamically allocated in the loop.

In actual Swiss tournaments, we also need to take care of possible re-matches, where 2 players may meet again in a later round, which is usually not allowed. However, we ignore this requirement in our assignment for simplicity.

Hint: You may look at the example described below to understand the Swiss algorithm.

End of Description

Resources & Sample Program

- Skeleton code: skeleton.zip
- Demo program (executable for Windows): pa1.exe
- Demo program (executable for Mac): pa1-mac.exe (you may need to change the file permission: `chmod 775 pa1-mac.exe`)

When you execute the program, the prompt will ask you to input the number of players, then each players' name and ELO rating, then number of rounds. As the matches are played, you can type numbers from 1 to 3 to input the result of each matches. Alternatively, you can pass a text file to the program as input:

```
pa1.exe < input/input1.txt
```

Note: On Windows, if your terminal is PowerShell, you may need to use the following command instead:

```
Get-Content input/input1.txt | .\pa1.exe
```

Some input examples have been provided for you in the skeleton code. Here is an example output with manual input, which is similar to **input/input3.txt**:

```
Please input number of players: 8

Please input each players' names followed by their ELO rating.
Alice
1000
Bob
1500
Carol
2000
David
2500
Ethan
1200
Fred
1450
Gary
2200
Harry
1750
Please input number of rounds: 3

==================================
ROUND 0/3 ENDED.
David (2500): 0
Gary (2200): 0
Carol (2000): 0
Harry (1750): 0
Bob (1500): 0
Fred (1450): 0
Ethan (1200): 0
Alice (1000): 0
==================================
David (2500) vs. Bob (1500)
1
David (2500) defeated Bob (1500)
Gary (2200) vs. Fred (1450)
2
Fred (1450) defeated Gary (2200)
Carol (2000) vs. Ethan (1200)
1
Carol (2000) defeated Ethan (1200)
Harry (1750) vs. Alice (1000)
3
Harry (1750) and Alice (1000) drew
==================================
ROUND 1/3 ENDED.
David (2500): 2
Carol (2000): 2
Fred (1450): 2
Harry (1750): 1
Alice (1000): 1
Gary (2200): 0
Bob (1500): 0
Ethan (1200): 0
==================================
David (2500) vs. Carol (2000)
3
David (2500) and Carol (2000) drew
Fred (1450) vs. Harry (1750)
1
Fred (1450) defeated Harry (1750)
Alice (1000) vs. Bob (1500)
2
Bob (1500) defeated Alice (1000)
```

```
Gary (2200) vs. Ethan (1200)
1
Gary (2200) defeated Ethan (1200)
==================================
ROUND 2/3 ENDED.
Fred (1450): 4
David (2500): 3
Carol (2000): 3
Gary (2200): 2
Bob (1500): 2
Harry (1750): 1
Alice (1000): 1
Ethan (1200): 0
==================================
Fred (1450) vs. David (2500)
2
David (2500) defeated Fred (1450)
Carol (2000) vs. Gary (2200)
3
Carol (2000) and Gary (2200) drew
Bob (1500) vs. Harry (1750)
1
Bob (1500) defeated Harry (1750)
Alice (1000) vs. Ethan (1200)
1
Alice (1000) defeated Ethan (1200)
==================================
ROUND 3/3 ENDED.
David (2500): 5
Carol (2000): 4
Bob (1500): 4
Fred (1450): 4
Gary (2200): 3
Alice (1000): 3
Harry (1750): 1
Ethan (1200): 0
==================================
```

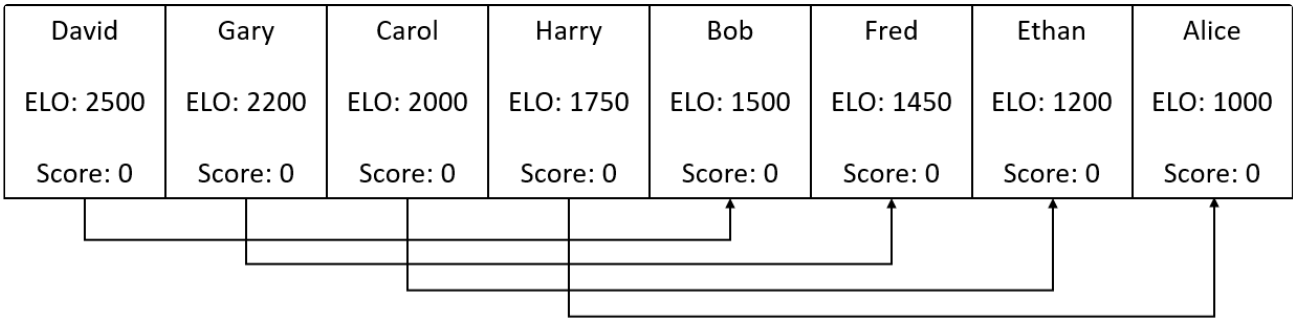
To demonstrate the algorithm described in Task 5, here is an illustration of the above tournament. The list of players begins with the 8 players:

Alice	Bob	Carol	David	Ethan	Fred	Gary	Harry
ELO: 1000	ELO: 1500	ELO: 2000	ELO: 2500	ELO: 1200	ELO: 1450	ELO: 2200	ELO: 1750
Score: 0	Score: 0	Score: 0	Score: 0	Score: 0	Score: 0	Score: 0	Score: 0

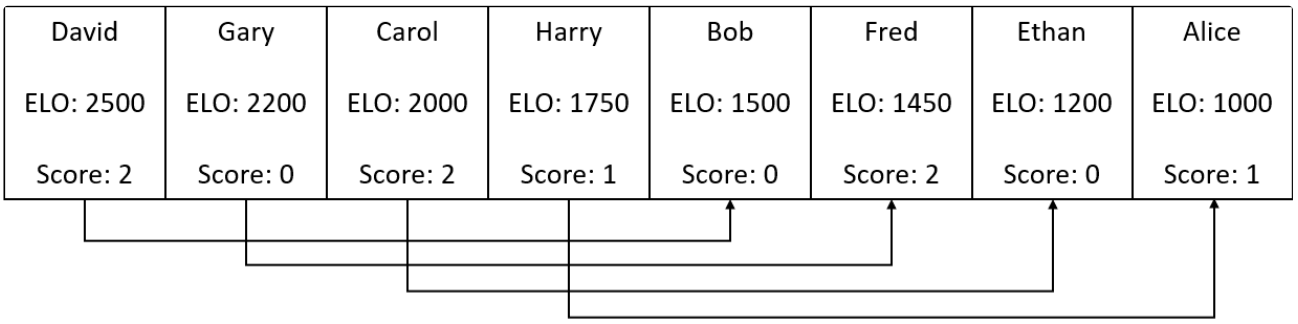
The `list` is sorted in the construction of the Swiss instance (seeding the players) - since all players have score 0, they are sorted by ELO:

David	Gary	Carol	Harry	Bob	Fred	Ethan	Alice
ELO: 2500	ELO: 2200	ELO: 2000	ELO: 1750	ELO: 1500	ELO: 1450	ELO: 1200	ELO: 1000
Score: 0	Score: 0	Score: 0	Score: 0	Score: 0	Score: 0	Score: 0	Score: 0

In the first round, all players have score 0. We then pair the top 4 players with the remaining 4 players:



The matches then get played, and the players' scores are updated:



At the end of round 1, we sort the `list` again to print the leaderboard:

End of Download

Testing

Since we no longer provide public test cases on ZINC this semester, you can perform testing on your local machine by compiling with the given Makefile. If all tasks have been implemented correctly, the generated program **pa1.exe** should have the same output as the demo program, given the same input. Provided in the skeleton code are also output text files corresponding to the input .txt files, which you can use to compare with your program.

In the actual grading, there will be more test cases designed to test each tasks separately. Therefore, even if you cannot finish all tasks, especially task 5, and are unable to compile a program with the expected output, you may still be able to get partial credits for the other tasks.

End of Testing

Submission and Deadline

Deadline: 23:59:00 on *March 19, 2023*.

ZINC Submission

Please submit the following files to ZINC by zipping the following 2 files. ZINC usage instructions can be found here.

```
player.cpp
swiss.cpp
```

Notes:

- The compiler used on ZINC is `g++11`. However, there shouldn't be any noticeable differences if you use other versions of the compiler to test on your local machine.

- **We will check for memory leak in the final grading**, so make sure your code does not have any memory leak.
- You may submit your file multiple times, but only the latest version will be graded.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day, especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we will grade your latest submission with all test cases after the deadline.
- If you have encountered any server-side problems or webpage glitches with ZINC, you may post on the ZINC support forum to get attention and help from the ZINC team quickly and directly. If you post on Piazza, you may not get the fastest response as we need to forward your report to them, and then forward their reply to you, etc.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online autograder ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts you cannot finish, just put in a dummy implementation so that your whole program can be compiled for ZINC to grade the other parts you have done. Empty implementations can be like:

```
int SomeClass::SomeFunctionICannotFinishRightNow()
{
  return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsGraded()
{
}
```

Reminders

Make sure you actually upload the correct version of your source files - we only grade what you upload. Some students in the past submitted an empty file or a wrong file or an exe file which is worth zero mark. So **you must double-check the file you have submitted**.

Late Submission Policy

There will be a penalty of -1 point (out of a maximum 100 points) for every minute you are late. For instance, since the deadline for assignment 2 is 23:59:00 on *March 19*, if you submit your solution at 1:00:00 on *March 20*, there will be a penalty of -61 points for your assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to late penalty (and any other penalties) will be reset to zero.

End of Submission and Deadline

Frequently Asked Questions

Q: The demo program does not work with the given .txt files.
A: If you downloaded an earlier version of the skeleton code, the input files may have the Windows ending sequence (CRLF), which will not work on Mac or Linux. You can use VS Code to change the ending sequence to Unix (LF), or download the latest version of the skeleton code.

Q: My code doesn't work / there is an error. Here is the code. Can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own, and we should not finish the tasks for you. We might provide some very general hints, but we shall not fix the problem or debug it for you.

Q: Can I add extra helper / global functions?
A: You may do so in the files that you are allowed to submit. That implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?
A: No, other than the mentioned libraries: **<cstring>** and **<iostream>**.

Q: Can I use global or static variables such as `static int x`?
A: No.

Q: Can I use `auto`?
A: No.

End of Frequently Asked Questions

Menu

Page maintained by

DINH Anh Dung
Email: dzung@ust.hk

Last Modified: 02/28/2023 09:36:50

Homepage

Course Homepage