## 1 Hints for Task A

Define `Profile` class. A profile is similar to a person, but is used in the world of network.

1. Data members are `username` and `displayname`. Since they are private, need to provide methods (aka operations) in public interfaces to access or modify these data members.

2. Constructors are used to initialize data members and create an object with needed operations – for example, getters or accessors like `getUsername`, `getFullName`, setters or mutators like `setDisplayName`.

```cpp
#include <iostream>
using namespace std;

class Profile {
private:
    string username;
    string displayname;
public:
    // Profile constructor for a user (initializing
    // private variables username=usrn, displayname=dspn)
    Profile(string usrn, string dspn);

    // Default Profile constructor (username="", displayname="")
    Profile();

    // Return username
    string getUsername();
    // Return name in the format: "displayname (@username)"
    string getFullName();
    // Change display name
    void setDisplayName(string dspn);
};

//TODO:
//(1) set data member username by formal parameter usrn,
//(2) set data member displayname by formal parameter dspn.
Profile::Profile(string usrn, string dspn) {


}

//TODO:
// Non-default Profile constructor
// (1) set data member username to be "",
// (2) set data member displayname to be "".
Profile::Profile() {


}

//TODO: Return username
string Profile::getUsername() {

}

//TODO: Return name in the format: "displayname (@username)"
string Profile::getFullName() {



}

//TODO: Change data member displayname by formal parameter dspn
void Profile::setDisplayName(string dspn) {

}

int main() {
    Profile p1("marco", "Marco");
    cout << p1.getUsername() << endl; // marco
    cout << p1.getFullName() << endl; // Marco (@marco)

    p1.setDisplayName("Marco Rossi");
    cout << p1.getUsername() << endl; // marco
    cout << p1.getFullName() << endl; // Marco Rossi (@marco)

    Profile p2("tarma1", "Tarma Roving");
    cout << p2.getUsername() << endl; // tarma1
    cout << p2.getFullName() << endl; // Tarma Roving (@tarma1)
}
```

Sample output is as follows.

```
marco
Marco (@marco)
marco
Marco Rossi (@marco)
tarma1
Tarma Roving (@tarma1)
```

## 2 Task B

Define class `Network`, describing how a profile in this network follows another. It has the following data members.

- `profiles` is an array of `Profile` whose capacity is `MAX_USERS` with value 20. That is, a network has at most twenty profiles.
- `MAX_USERS` is a **static data member**, also called **class data member**. Think each network has exactly the same value for this data member, or, this data member is shared by all objects of `Network`. So, we need only one copy of this data member in the whole class.

A similar example is Person class. Suppose we only care about name, age, and the number of total population in the world. Each person has name and age, which differ from one individual person to another. However. the number of total population in the world should be set as static data member since this value is not changed from one person to another.

- `numUsers` stores the current size (number of profiles) of data member `profiles` in the network.
- Note that we use a static allocated array of profiles, so the size is a constant – `MAX_USERS` – and we use `numUsers` to specify the size of the array, or the actual number of elements in array `profiles`.
- Here is a sidenote describing the difference between static allocated array and dynamically allocated array. In static allocated array, the capacity (maximum number of elements allowed) must be a const. There is only one chance in compilation time to apply memory for the array, so we normally apply for the maximum number of elements we need for the application, aka, capacity. However, we may not use all the capacity of the array, so we need to use another variable called size to track the current number of elements in the array.

  It is like when you apply to build a hotel, you would apply for the maximum of rooms allowed, that is call `capacity`, this needs to be a const. But the actual occupied rooms can be vary and is saved in `size`.

  In contrast, the size of a dynamically allocated array does not need to be const. However, we need to use `new` to apply for memory and `delete` to release memory when we no longer need it.

  In Project Minesweeper, we use dynamically allocate one-dimensional array `int* cells` whose `size` can be determined in run time. It is like dynamically allocated array is tailored and fit to the need exactly, no unused elements in the array.

  Operator * in `int*` implies one or many. Variable of type `int*` can save the address of one int, it can also save the initial address of an array of ints. In Project Minesweeper, `cells` stores the initial address (ie, the address of the first element) in an array of ints.

  In Project Game 1024, we use dynamically allocated two-dimensional array `int** panel`. Type `int**` saves the address of an array of `int*`, the size of this array is `numRows`, and each `int*` saves the address of an array of `int`, the size of that array is `numCols`.

  Said differently, `panel` has `numRows` rows, each row points to (save the initial address) of an array of `numCols` ints. So `panel` is a two-dimensional array of ints.

```cpp
#include <iostream>
using namespace std;

class Profile {
private:
    string username;
    string displayname;
public:
    // Profile constructor for a user (initializing
    // private variables username=usrn, displayname=dspn)
    Profile(string usrn, string dspn);

    // Default Profile constructor (username="", displayname="")
    Profile();

    // Return username
    string getUsername();

    // Return name in the format: "displayname (@username)"
    string getFullName();

    // Change display name
    void setDisplayName(string dspn);
};

//TODO:
// Non-default Profile constructor
// (1) set data member username to be "",
// (2) set data member displayname to be "".
Profile::Profile(string usrn, string dspn) {


}

//TODO: Default Profile constructor (username="", displayname="")
Profile::Profile() {

}

//TODO: Return username
string Profile::getUsername() {

}

//TODO: Return name in the format: "displayname (@username)"
string Profile::getFullName() {



}

//TODO: Change display name
void Profile::setDisplayName(string dspn) {

}

class Network {
private:
  static const int MAX_USERS = 20; // max number of user profiles
  int numUsers;                    // number of registered users
  Profile profiles[MAX_USERS];     // user profiles array:
                                   // mapping integer ID -> Profile

  // Returns user ID (index in the 'profiles' array) by their username
  // (or -1 if username is not found)
  int findID (string usrn);

public:
  // Constructor, makes an empty network (numUsers = 0)
  Network();

  // Attempts to sign up a new user with specified username and displayname
  // return true if the operation was successful, otherwise return false
  bool addUser(string usrn, string dspn);
```

```
};

//TODO: set data member numUsers to be zero.
Network::Network() {

}

//TODO: find out whether usrn is in the profiles or not.
//If yes, return the index, otherwise, return -1.
int Network::findID(string usrn) {





}

//TODO: Attempts to sign up a new user with specified username and displayname
// return true if the operation was successful, otherwise return false
//The specified usrn needs to contain alphabet and digit letter only. That is, only 'A'-'Z', 'a'-'z',
    or '0'-'9' are allowed.
//Hint: use int isalpha ( int c ) to test
//whether character c is alphabetic, ie, 'a'-'z', or 'A'-'Z'.
//Similarly, use int isdigit ( int c ) to test
//whether character c is decimal digit '0' - '9'.
//To use isalpha or isdigit, need to
//include cctype libary by
//#include <cctype>
bool Network::addUser(string usrn, string dspn) {
    //TODO: if any character in usrn is
    //neither alphabet nor digit, return false


    //TODO: if numUsers is larger than equal to MAX_USERS,
    //return false.


    //TODO: if usrn is not an exisiting username,
    //ie, usrn not found in profiles,
    //add usr to profiles, increase numUsers by one.
    //
    //hints: call findID method of Network class with usrn.
    //If the return is -1, then usrn is not found in profiles.
    //create a profile object with formal parameters
    //usrn and dspn,
    //and put that profile object to data member profiles,
    //increase number of users, denoted by numUsers, by 1.
    //Do not forget to return true.
    if (findID(usrn) == -1) { //usrn not found in profiles



    }
    else //usrn is an existing user in profiles.
         //TODO: what is the return?
```

```

}

int main() {
    Network nw;
    cout << nw.addUser("mario", "Mario") << endl; // true (1)
    cout << nw.addUser("luigi", "Luigi") << endl; // true (1)

    cout << nw.addUser("mario", "Mario2") << endl; // false (0)
    cout << nw.addUser("mario 2", "Mario2") << endl; // false (0)
    cout << nw.addUser("mario-2", "Mario2") << endl; // false (0)

    for(int i = 2; i < 20; i++)
        cout << nw.addUser("mario" + to_string(i),
                   "Mario" + to_string(i)) << endl; // true (1)

    cout << nw.addUser("yoshi", "Yoshi") << endl; // false (0)
}
```

Sample output is as follows.

```
1
1
0
0
0
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
0
```

## 3 Task C

In additional to the code of Task B, define two more methods `follow` and `printDot` of `Network` class.

```
#include <iostream>
using namespace std;
```

```

class Profile {
private:
    string username;
    string displayname;
public:
    // Profile constructor for a user (initializing
    // private variables username=usrn, displayname=dspn)
    Profile(string usrn, string dspn);

    // Default Profile constructor (username="", displayname="")
    Profile();

    // Return username
    string getUsername();

    // Return name in the format: "displayname (@username)"
    string getFullName();

    // Change display name
    void setDisplayName(string dspn);
};

//TODO:
// Non-default Profile constructor
// (1) set data member username to be "",
// (2) set data member displayname to be "".
Profile::Profile(string usrn, string dspn) {


}

//TODO: Default Profile constructor (username="", displayname="")
Profile::Profile() {

}

//TODO: Return username
string Profile::getUsername() {

}

//TODO: Return name in the format: "displayname (@username)"
string Profile::getFullName() {



}

//TODO: Change display name
void Profile::setDisplayName(string dspn) {

}

class Network {
```

```
private:
  static const int MAX_USERS = 20; // max number of user profiles
  int numUsers;                    // number of registered users
  Profile profiles[MAX_USERS];  // user profiles array:
                                   // mapping integer ID -> Profile

  // Returns user ID (index in the 'profiles' array) by their username
  // (or -1 if username is not found)
  int findID (string usrn);

  bool following[MAX_USERS][MAX_USERS]; // friendship matrix:
  // following[id1][id2] == true when id1 is following id2

public:
  // Constructor, makes an empty network (numUsers = 0)
  Network();
  // Attempts to sign up a new user with specified username and displayname
  // return true if the operation was successful, otherwise return false
  bool addUser(string usrn, string dspn);

  // Make 'usrn1' follow 'usrn2' (if both usernames are in the network).
 // return true if success (if both usernames exist), otherwise return false
 bool follow(string usrn1, string usrn2);

 // Print Dot file (graphical representation of the network)
 void printDot();
};

Network::Network() {
    //TODO: set data member numUsers to be zero


    //no one user is following another user yet.
    //That is, every element in following is false.

}

//TODO: find out whether usrn is in the profiles or not.
//If yes, return the index, otherwise, return -1.
int Network::findID(string usrn) {





}

//TODO: Attempts to sign up a new user with specified username and displayname
// return true if the operation was successful, otherwise return false
//The specified usrn needs to contain alphabet and digit letter only. That is, only 'A'-'Z', 'a'-'z',
    or '0'-'9' are allowed.
//Hint: use  int isalpha ( int c )  to test
//whether character c is alphabetic, ie, 'a'-'z', or 'A'-'Z'.
//Similarly, use  int isdigit ( int c )  to test
//whether character c is decimal digit '0' - '9'.
```

```
//To use  isalpha  or  isdigit ,
//need to include cctype libary.
//#include <cctype>
bool Network::addUser(string usrn, string dspn) {
    //TODO: if any character in usrn is
    //neither alphabet nor digit, return false


    //TODO: if numUsers is larger than equal to MAX_USERS,
    //return false.


    //TODO: if usrn is not an exisiting username,
    //ie, usrn not found in profiles,
    //add usr to profiles, increase numUsers by one.
    //
    //hints: call findID method of Network class with usrn.
    //If the return is -1, then usrn is not found in profiles.
    //otherwise, put usrn, dspn profile object to
    //data member profiles,
    //increase number of users, denoted by numUsers, by 1.
    //Do not forget to return true.
    if (findID(usrn) == -1) { //usrn not found in profiles



    }
    else //usrn is an existing user in profiles.
         //TODO: what is the return?

}

//Find out whether usrn1 follows usrn2.
//Hint: use findID to see whether usrn1 or usrn2 exists or not.
//Put the result of findID on usrn1 to id1,
//and the result of findID on usrn2 to id2,
//if both usrn1 and usrn2 exist,
//ie, neither id1 nor id2 is -1,
//set following[id1][id2] to be true and then return true;
//otherwise, at least one of id1 and id2 is -1,
//set following[id1][id2] to be false and
//return false.
bool Network::follow(string usrn1, string usrn2) {



}

//TODO: Print Dot file (graphical representation of the network).
//Hints:
//(1) To print double quotes " symbol,
//   use cout << "\"";
//(2) Use a nested loop to find the followers of each user.
void Network::printDot() {

}

int main() {
    Network nw;
    // add three users
    nw.addUser("mario", "Mario");
    nw.addUser("luigi", "Luigi");
    nw.addUser("yoshi", "Yoshi");

    // make them follow each other
    nw.follow("mario", "luigi");
    nw.follow("mario", "yoshi");
    nw.follow("luigi", "mario");
    nw.follow("luigi", "yoshi");
    nw.follow("yoshi", "mario");
    nw.follow("yoshi", "luigi");

    // add a user who does not follow others
    nw.addUser("wario", "Wario");

    // add clone users who follow @mario
    for(int i = 2; i < 6; i++) {
        string usrn = "mario" + to_string(i);
        string dspn = "Mario " + to_string(i);
        nw.addUser(usrn, dspn);
        nw.follow(usrn, "mario");
    }
    // additionally, make @mario2 follow @luigi
    nw.follow("mario2", "luigi");

    nw.printDot();
}
```

Sample output is as follows.

```
digraph {
"@mario"
"@luigi"
"@yoshi"
"@wario"
"@mario2"
"@mario3"
"@mario4"
"@mario5"
"@mario"->"@luigi"
"@mario"->"@yoshi"
"@luigi"->"@mario"
"@luigi"->"@yoshi"
"@yoshi"->"@mario"
"@yoshi"->"@luigi"
"@mario2"->"@mario"
"@mario2"->"@luigi"
"@mario3"->"@mario"
"@mario4"->"@mario"
"@mario5"->"@mario"
}
```

## 4 Task D

In addtional to codes in Task C, do the following:

- Define struct `Point`. A struct is a simplified class with all data members are public and no methods operating on those data members.
- Define methods `writePost` and `printTimeline` for class `Network`.

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Profile {
private:
    string username;
    string displayname;
public:
    // Profile constructor for a user (initializing
    // private variables username=usrn, displayname=dspn)
    Profile(string usrn, string dspn);

    // Default Profile constructor (username="", displayname="")
    Profile();

    // Return username
    string getUsername();

    // Return name in the format: "displayname (@username)"
    string getFullName();

    // Change display name
    void setDisplayName(string dspn);
};

Profile::Profile(string usrn, string dspn) {

}

//TODO: Default Profile constructor (username="", displayname="")
Profile::Profile() : Profile("", "") {
}

//TODO: Return username
string Profile::getUsername() {

}

//TODO: Return name in the format: "displayname (@username)"
string Profile::getFullName() {

}

//TODO: Change display name
void Profile::setDisplayName(string dspn) {

}

//ADDED in Task D
struct Post{
  string username;
  string message;
};

class Network {
private:
  static const int MAX_USERS = 20; // max number of user profiles
  int numUsers;                    // number of registered users
  Profile profiles[MAX_USERS];     // user profiles array:
                                   // mapping integer ID -> Profile

  // Returns user ID (index in the 'profiles' array) by their username
  // (or -1 if username is not found)
  int findID (string usrn);

  bool following[MAX_USERS][MAX_USERS]; // friendship matrix:
 // following[id1][id2] == true when id1 is following id2

  static const int MAX_POSTS = 100;
  int numPosts;                    // number of posts
  Post posts[MAX_POSTS];           // array of all posts

public:
  // Constructor, makes an empty network (numUsers = 0)
  Network();

  // Attempts to sign up a new user with specified username and displayname
  // return true if the operation was successful, otherwise return false
  bool addUser(string usrn, string dspn);

  // Make 'usrn1' follow 'usrn2' (if both usernames are in the network).
 // return true if success (if both usernames exist), otherwise return false
 bool follow(string usrn1, string usrn2);

 // Print Dot file (graphical representation of the network)
 void printDot();
 bool writePost(string usrn, string msg); // new
 bool printTimeline(string usrn);         // new
};

Network::Network() {
    //Keep the code in Task C.

    //TODO: set numPosts to be zero.
    //needed, or numPosts is not initialized
```

```
}

//Same as the code in Task C.
int Network::findID(string usrn) {

}

//Same as the code in Task C.
bool Network::addUser(string usrn, string dspn) {

}

//Same as the code in Task C.
bool Network::follow(string usrn1, string usrn2) {

}

//Same as the code in Task C.
void Network::printDot() {

}

//TODO: add in Task D.
bool Network::writePost(string usrn, string msg) {
    //if numPosts is larger than or equal to MAX_POSTS,
    //return false.
    //Data member numPosts is the index
    //of next post to write,
    //it cannot be equal to MAX_POSTS,
    //let alone to be larger than MAX_POSTS.


    //Save usrn and msg to the corresponding attribute of
    //the (numPosts)th element of posts.


    //Increase numPosts by 1.

    //Now a post with info of usrn and msg is
    //successfully put to array posts,
    //what should we return?

}

//TODO: newly added method of Task D.
bool Network::printTimeline(string usrn) {
    //Key steps:
    //(1) Check that usrn exists in profiles using findID,
    //    if not, return false.
    //    Put the return in variable id.
    //(2) Display posts in reverse-chronological order.
    //    Hints: read hints for Task D of Lab 3 in
    //https://tong-yee.github.io/135/lab_hints/hints_lab3_f23.pdf.
    //(3) Display all the posts of usrn or his/her followers,
    //    check whether username of the current post is a follower
    //   of usrn or not. Find id of username and put in id2.
    //   Check following[id][id2] is true or not.

}

int main() {
    Network nw;
    // add three users
    nw.addUser("mario", "Mario");
    nw.addUser("luigi", "Luigi");
    nw.addUser("yoshi", "Yoshi");

    nw.follow("mario", "luigi");
    nw.follow("luigi", "mario");
    nw.follow("luigi", "yoshi");
    nw.follow("yoshi", "mario");

    // write some posts
    nw.writePost("mario", "It's a-me, Mario!");
    nw.writePost("luigi", "Hey hey!");
    nw.writePost("mario", "Hi Luigi!");
    nw.writePost("yoshi", "Test 1");
    nw.writePost("yoshi", "Test 2");
    nw.writePost("luigi", "I just hope this crazy plan of yours works!");
    nw.writePost("mario", "My crazy plans always work!");
    nw.writePost("yoshi", "Test 3");
    nw.writePost("yoshi", "Test 4");
    nw.writePost("yoshi", "Test 5");

    cout << endl;
    cout << "======= Mario's timeline =======" << endl;
    nw.printTimeline("mario");
    cout << endl;

    cout << "======= Yoshi's timeline =======" << endl;
    nw.printTimeline("yoshi");
    cout << endl;
}
```

Sample output is as follows.

```
======= Mario's timeline =======
Mario (@mario) My crazy plans always work!
Luigi (@luigi) I just hope this crazy plan of yours works!
Mario (@mario) Hi Luigi!
Luigi (@luigi) Hey hey!
Mario (@mario) It's a-me, Mario!

======= Yoshi's timeline =======
Yoshi (@yoshi) Test 5
Yoshi (@yoshi) Test 4
Yoshi (@yoshi) Test 3
Mario (@mario) My crazy plans always work!
Yoshi (@yoshi) Test 2
Yoshi (@yoshi) Test 1
Mario (@mario) Hi Luigi!
Mario (@mario) It’s a-me, Mario!
```