Lab 13. Classes and Programming a Social Network

Introducing Object-Oriented Programming

Object-oriented programming (OOP) is a paradigm of software design based on the concept of "objects", which may contain data (variables), and code (functions, often known as methods).

Units of OOP design are called **objects**, they are grouping together variables and functions. In C++, objects are implemented using **classes**.

In C++, **classes** are compound datatypes similar to `struct`, which contain:

- data fields also called **member variables**,
- **member functions**, which can operate on the member variables,
- **constructors** and **destructors** (special functions called when instances of a class get created and deleted).

Member variables and member functions can be declared `public` or `private`, thus making them accessible or not accessible from outside the class. This principle is called **encapsulation**, classes designed according to this principle, separate their interface from their implementation:

- **Public** functions provide the public interface, and can be called by other classes.
- **Private** functions can be used only internally by the class itself, and constitute the implementation details of the class. (Member variables are usually declared private and so cannot be directly modified or accessed from outside.)

A well defined class provides a concise interface that *does not depend* on the specific details of its implementation. A user of a class is generally not expected to know and should not worry about the specifics of the class implementation. Since the class interacts with the external world only via its public interface, its implementation becomes abstracted away (it does not matter to the user, how the class is implemented).

Instances of classes are called **objects**. In simple terms, classes are types, and objects are variables.

Example: Declaration of a class Particle

```
class Particle {
private:
  double x;  // position, 2D coordinates
  double y;
  double vx; // velocity
  double vy;
public:
  // accessor functions for the current position of the particle
  double getX();
  double getY();
  // move the particle
  void move(double dt);
  // Constructor. Called to initialize private member variables
  // when a particle object gets created
  Particle(double posx, double posy, double velx, double vely);
  // Default constructor, will assume pos=(0,0), vel=(0,0)
  Particle();
};
```

Its usage in main function

```
int main() {
  // Create a particle object using the constructor
  Particle particle1 (11.0, 12.0,  0.5, -0.4);
  // Create another particle object, but this time
  // using the default constructor
  Particle particle2;
```

```
  particle1.move(10.0);
  cout << particle1.getX() << endl;
  cout << particle1.getY() << endl;

  cout << particle1.x << endl;       // Compile-time ERROR,
                                     // cannot access
                                     // private variable
  particle1.x += particle1.vx * 5.0; // same ERROR
}
```

Implementation of the class functions and constructors

```
double Particle::getX() {
  return x;
}
double Particle::getY() {
  return y;
}
void Particle::move(double dt) {
  x += vx * dt;  y += vy * dt;
}
Particle::Particle(double posx, double posy, double velx, double vely) {
  x = posx;   y = posy;
  vx = velx;  vy = vely;
}
Particle::Particle() {
  x = 0;   y = 0;
  vx = 0;  vy = 0;
}
```

A remark on using member initializer list when defining constructors

There exist another way for initializing variables in constructors, called **member initializer list**. The above two constructors can be defined as follows:

```
Particle::Particle(double posx, double posy, double velx, double vely)
  : x(posx), y(posy), vx(velx), vy(vely) { }

Particle::Particle() : x(0), y(0), vx(0), vy(0) { }
```

When the initialized variables are objects, this initialization method can be more efficient than the assignment operations, which we used in the first implementation. You can use both approaches in

your code, they are both acceptable C++.

A complete Particle program is available here: http://codepad.org/Qd422fMA. One of its constructors is defined using an initializer list, and the other with simple assignments.

Designing a simple social network program

Class `Profile`

Represents the information about each user, each user profile has:

- a *username*, a non-empty alphanumeric string that uniquely identifies the user.
- a *display name*, which can be an arbitrary string.

It provides an interface to:

- Return the current display name and username.
- Change the display name (but the username cannot be changed).

Class `Network`

It stores the information about the entire network. It has:

- an array with all registered user profiles.
- a 2D array to remember who is following who.
- an array with all user posts.

It provides an interface to:

- add new user
- make one user follow another user
- print the diagram of the network
- posting
- printing the timeline of a user

Task A. Class `Profile`

In this task, write a program `social.cpp`, in which you have to implement the class `Profile` that can store the info about a user of the network.

It has **two private properties** of type string: *displayname* and *username*.

The **public interface** consists of two constructors initializing the private variables and three functions:
`getUsername` returns the username.
`getFullName` returns the string in the format “displayname (@username)”.
`setDisplayname` allows to change the displayname property. (Username is a unique user identifier and cannot change, while displayname can be any string and can be modified, so the class provides a mechanism for updating it.)

The full class declaration is shown below:

```
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);
};
```

A usage example
(*When submitting your code, please use this exact main function*):

```
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)
}
```

Task B. Class `Network` : Adding users

The three main functionalities of the class `Network` are: 1) adding new users to the network, 2) following, and 3) posting messages. We are going to develop them in these three tasks.

Write a program `social2.cpp`, implementing the first version of the class `Network` (feel free to copy your class `Profile` and anything else from the previous program).

This first version of the class should be able to add new users and store their profiles, for that, the class has the following **private variables**:

- an array of registered user `profiles`,
- an integer `numUsers`, which stores the number of registered users.
- the size of the `profiles` array is defined as `MAX_USERS=20`, we cannot sign up more than this number of users.

The index in the array `profile` is the **profile ID**. As you keep adding new users, they are receiving ID = 0, 1, 2, and so on. To look up the ID of a specific profile by its `username`, we have a private function `findID(usrn)`, it should return the ID of the user with that username, or -1 if the user is not found in the profiles array.

The **public interface**:

- a default constructor that creates an empty network. For now, setting `numUsers = 0` is enough.
- `addUser(usrn, dspn)` is the central function for the class. It allows adding new users. This function receives the username and display name of a new user, the new user can be signed up only if the following conditions are met:

- the new username `usrn` must be a *non-empty alphanumeric* string,
- there is no other users in the network with the same username,
- the array `profiles` is not full.

If the above conditions are met, the new user should be added to the array `profiles` and the function should return `true`, otherwise don't add the user and return `false`, thus indicating that the operation failed.

The full class declaration is shown below:

```
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);
};
```

A usage example (it needs `--std=c++11` flag to compile)
(*When submitting your code, please use this exact main function*):

```
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)
}
```

Task C. Class `Network` : Following others

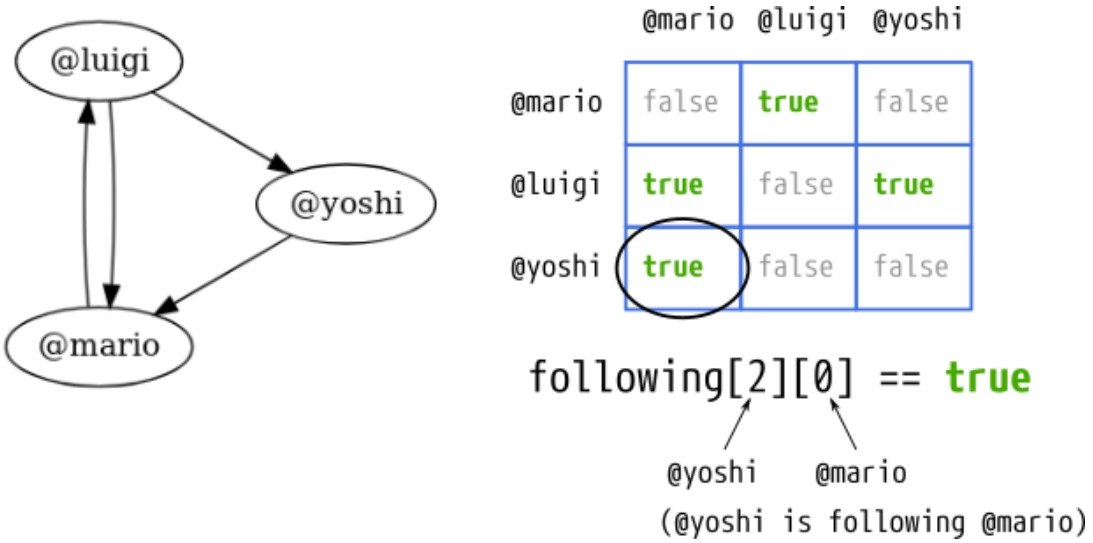
Write a new program `social3.cpp`, which is an improved version of the previous program.

The class `Network` should be changed to keep the friendship (following) relation between the users. You should:

1. add one **private variable**:

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

   An example demonstrating this friendship matrix for a small network of three users is shown below:



2. The constructor `Network()` should be modified, setting all elements of the matrix `following` to `false`, so in empty network no one is following anyone.

3. Add two functions to the **public interface**:

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

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

The first new function `follow(usrn1, usrn2)` simply marks the corresponding cell in the friendship matrix if both usernames are found. It returns `true` or `false`, depending on whether the operation was successful or not.

The second new function `printDot()` will be used to visualize the network. For that, make the function print out the network in the following format:

```
digraph {
  "@mario"
  "@luigi"
  "@yoshi"

  "@mario" -> "@luigi"
  "@luigi" -> "@mario"
  "@luigi" -> "@yoshi"
  "@yoshi" -> "@mario"
}
```

Here, we are listing usernames with @ and in quotes, followed by each of the friendship connections, each on its own line. The example above is the exact output the function should produce for the simple network of three users @mario, @luigi, and @yoshi provided previously.

This output format is called “Dot file format”, it can be used in the application Graphviz to generate a visual diagram of the network. We can use this *Online Graphviz app*: https://dreampuf.github.io/GraphvizOnline/. Paste the Dot output in the left pane, and you will get a picture in the right pane.

To summarize, the new modified class declaration should look as follows:

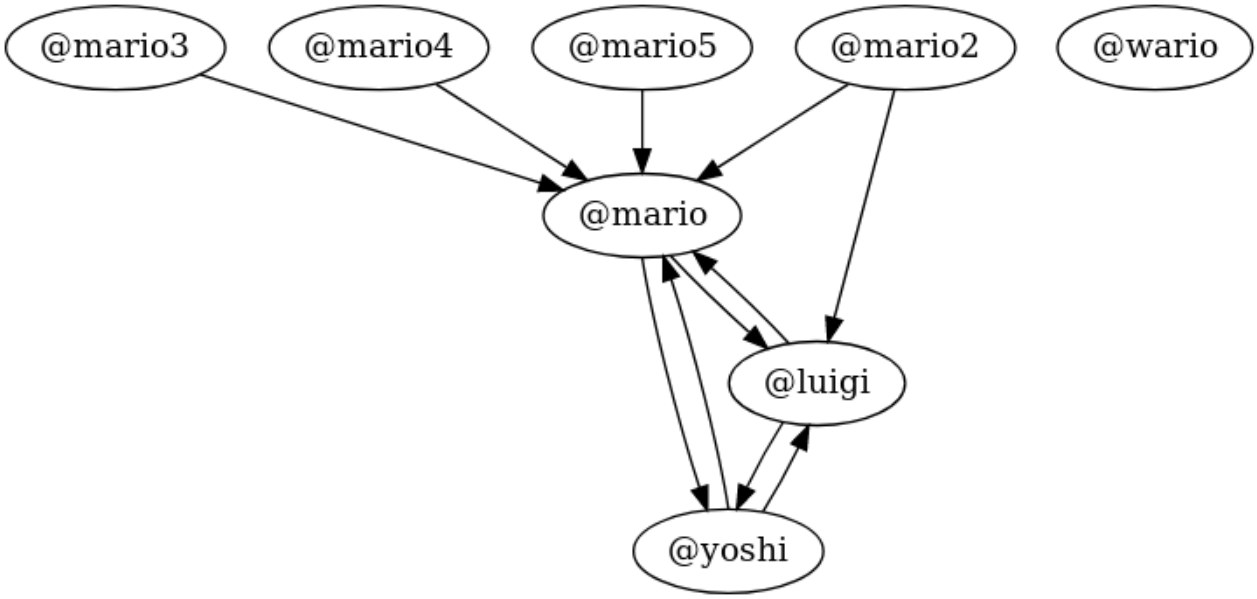
```
class Network {
private:
    static const int MAX_USERS = 20;
    int numUsers;
    Profile profiles[MAX_USERS];
    bool following[MAX_USERS][MAX_USERS];    // new
    int findID (string usrn);
public:
    Network();
```

```
    bool addUser(string usrn, string dspn);
    bool follow(string usrn1, string usrn2); // new
    void printDot();                         // new
};
```

A usage example (it needs `--std=c++11` flag to compile)
(*When submitting your code, please use this exact main function*):

```
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();
}
```

A Graphviz visualization of the network produced by the example above:



Task D. Class `Network` : Posting (Bonus)

Write a new program `social4.cpp`, which is an improved version of the previous program.

The class `Network` should be changed to allow users post messages and remember them. You should:

1. (Outside the class `Network`), define a new struct:

```
struct Post{
  string username;
  string message;
};
```

2. In the class `Network`, add **private variables**:

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

3. Modify constructor `Network` so that it initializes `numPosts` to 0.

4. Add two functions in the **public interface** of the class:

```
public:
 // Add a new post
 bool writePost(string usrn, string msg);
```

```
    // Print user's "timeline"
    bool printTimeline(string usrn);
```

The first function, `writePost(usrn, msg)` adds a new post to the `posts` array. It performs successfully if the username is found in the network and the `posts` array is not full, in this case the function also should return `true`. Otherwise, nothing is added and the function returns `false`.

The second function, `printTimeline(usrn)` prints out the timeline of the user `usrn`. The **timeline** of a user is the list of **all posts by the user and by the people they follow**, presented in *reverse-chronological* order. They should be printed in the following format:

```
Displayname (@username): message
Displayname (@username): message
Displayname (@username): message
Displayname (@username): message
```

To summarize, the final version of the class `Network` and struct `Post` declarations:

```
struct Post{
    string username;
    string message;
};

class Network {
private:
    static const int MAX_USERS = 20;
    int numUsers;
    Profile profiles[MAX_USERS];
    bool following[MAX_USERS][MAX_USERS];
    static const int MAX_POSTS = 100;          // new
    int numPosts;                              // new
    Post posts[MAX_POSTS];                     // new
    int findID (string usrn);
public:
    Network();
    bool addUser(string usrn, string dspn);
    bool follow(string usrn1, string usrn2);
    void printDot();
    bool writePost(string usrn, string msg);   // new
    bool printTimeline(string usrn);           // new
};
```

A usage example

(*When submitting your code, please use this exact main function*):

```
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;
}
```

It generates a small network of three users and prints out the timelines of two of them:

```
======= 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!
```

How to submit your programs

Each program should be submitted through Gradescope

Write separate programs for each part of the assignment.
Submit only the source code (.cpp) files, not the compiled executables.
Each program should start with a comment that contains your name and a short program description, for example:

```
/*
Author: your name
Course: CSCI-136
Instructor: their name
Assignment: title, e.g., Lab1A

Here, briefly, at least in one or a few sentences
describe what the program does.
*/
```