Classes and Programming a Social Network

Lab 11: https://maryash.github.io/135/labs/lab_11.html

- Sadab Hafiz

Introduction to 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 functions.

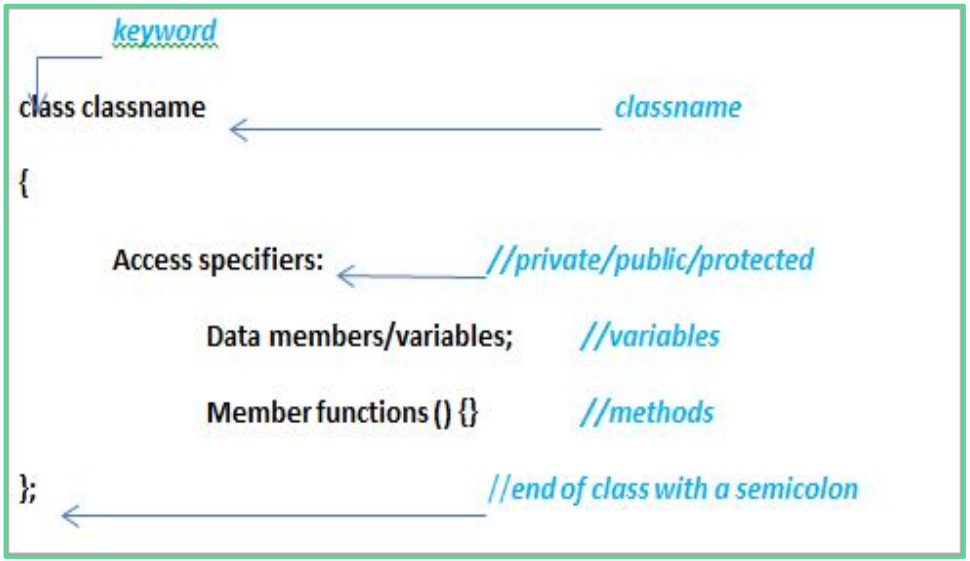
Units of OOP design are called **objects**, they are grouping together variables and functions. In C++, objects are implemented using **classes**. Instances of classes are called **objects**. In simple terms, **classes** are types, and **objects** are variables.

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).

Introduction to Classes

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

- data members, also referred to as fields
- member functions, which can operate on the data members
- constructors and destructors (special functions called when instances of a class get created and deleted)



Data members 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.
Access specifiers allow encapsulation.

Access Specifiers (recap)

Access Specifiers are used to specify who can access certain data members and member functions of a class:

public: Can be accessed by anyone who has access to an object of this class. Usually there are getters and setter functions which allows users to modify private fields. More things are usually public including constructors, destructors etc. that are covered in CS235.

private: Can be accessed ONLY by the class itself. Usually sensitive data members and functions are specified as private. Most classes have their data members as private.

protected: Can be accessed by classes that inherit from the current class. This will be covered more in CS235 and maybe later this semester.

Example class provided in Lab 11

```
class Particle {

	private:

		double x;                                                    // private data member

		double y;                                                    // private data member

		double vx;                                                   // private data member

		double vy;                                                   // private data member

	public:

		double getX();                                               // getter function that returns x

		double getY();                                               // getter function that returns y

		void move(double dt);                                        // public member function

		Particle(double posx, double posy, double velx, double vely);  // parameterized constructor

		Particle();                                                  // default constructor

};
```

You should take a look at the entire implementation from the lab.

Introduction to Constructors

Constructors are functions that are called by a class when we try creating an object of that particular class. A class can have multiple constructors. A constructor without any parameter is called the default constructor. Constructors are defined with the same name as the class:

```
class Particle {

       . . . other data members and functions are same as the previous slide

       Particle(double posx, double posy, double velx, double vely);        //parameterized constructor

       Particle();                                                          //default constructor

};

Particle::Particle(double posx, double posy, double velx, double vely) {    //parameterized constructor implementation

       x = posx;   y = posy;   vx = velx;   vy = vely;

}

Particle::Particle() {                                                       //default constructor implementation

       x = 0;   y = 0;   vx = 0;   vy = 0;

}
```

Default vs Parameterized

The default constructor is invoked when we create a class without giving it any parameters or value. The parameterized constructor is called when creating a class with some parameters. Constructors allow complex logic that during object initialization.

```
. . . look at the constructor definition and implementation from the last two slides

int main(){

	Particle default_particle;                                  // default constructor

	Particle parameterized_particle(0.1, 0.2, 0.3, 0.4);        // parameterized constructor

	cout << default_particle.x << endl;                         // will print 0 or will it?

	cout << parameterized_particle.x << endl;                   // will print 0.1 or will it?

}
```

Constructors are always public because you want users of your class to use the constructors to create objects of that class. Constructors also don’t return anything.

Member initializer list

There exists another way for initializing data members in constructors, called **member initializer list**. The two constructors of Particle class can be defined as follows:

```
// parameterized constructor

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

// default constructor

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 in C++.

The lab provides a link to an entire Particle program that shows the use of both member initializer list and the regular way of implementing constructors.

Designing a simple Social Network program

We can represent a social network using a Profile class and a Network class.

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: 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

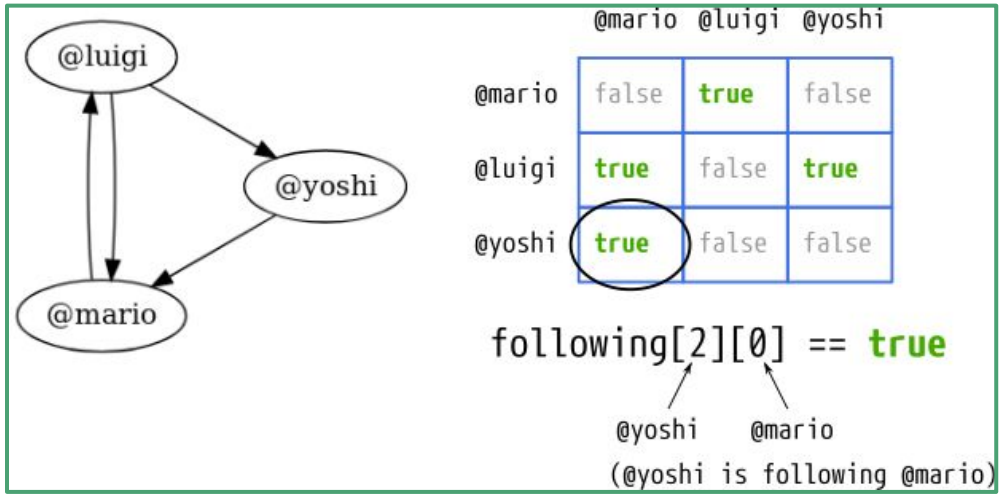
It provides an interface to:

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

Adding the 2D boolean array (Task C)

bool following[MAX_USERS][MAX_USERS];

Each coordinate of this 2D array represents if one user is following another user. For example, if following[0][1] is true, then user with ID 0 is following user with ID 1. In other words, profiles[0] is following profiles[1].



All the coordinates of the 2D array needs to be false in the beginning. Thus, the Network() constructor has to initialize the 2D array with all false values since boolean arrays are not autoinitialized.

Pseudocode for follow() function (Task C)

bool follow(string usrn1, string usrn2)

Calling this function will make profile with username usrn1 follow the profile with username usrn2. The function will return false if any of the two users are not found in the Network. Something like this:

```
// everything in green is pseudocode

bool Network :: follow(string usrn1, string usrn2){

       int usrn1_id = . . . get the id of usrn1 using findID function from Task B ;

       int usrn2_id = . . . get the id of usrn2 using findID function from Task B ;

       . . . return false if either usrn1_id or usrn2_id is equal to -1 (not in the network)

       . . . make user with usrn1_id follow user with usrn2_id (set the corresponding value in 2D array equal to true)

       return true;

}
```

The user ID corresponds to the **index** of the user in the 2D matrix.

Implementing printDot() function

void printDot()
This function prints the following array in a specific format provided in the lab. This format is known as “Dot file format”.

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

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

The quotes here are also to be printed. The lines in the middle have two leading spaces. You can print a quote like this: cout << “\“” ;

The first half of this output lists all the usernames of the profiles currently in the Network.

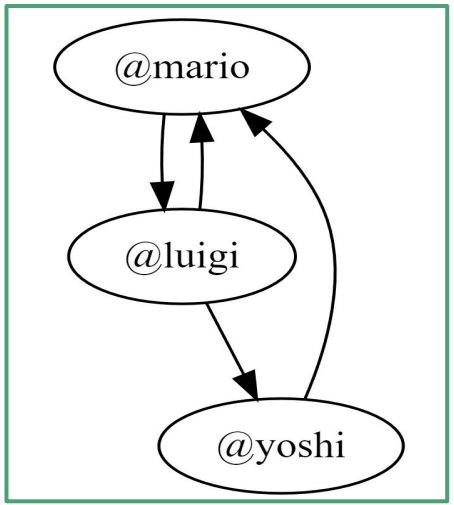
The second part of the output prints “@user1” -> “@user2” where Profile with username user1 follows Profile with username user2.

Why this output?

The output for printDot() function can be used in an application known as GraphViz to generate a diagram showing a visual of the Network. You can paste the output in this link: Graphviz Online

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

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



Pseudocode for printDot() function

void printDot()

First and last line should be simple print statements using cout. Let's break the rest of the output into multiple parts:

```
"@mario"
"@luigi"
"@yoshi"
```

For this part, we have to print all the usernames in the Network in this format. Iterate through profiles array using a for loop(variable i). You can print a single user like this:

```
cout<<"  \"@"<<profiles[i].getUsername()<<"\""<<endl;
```

DON'T FORGET THE ENDLINE(endl or '\n') BETWEEN THE TWO PARTS

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

Use a nested for loop(i, j) to iterate the following 2D array. While iterating it, if following[i][j] == true, we can print a single line of user followed relationship like this:

```
cout<<"  \"@"<<profiles[i].getUsername()<<"\" -> \"@"<<
profiles[j].getUsername()<<"\""<<endl;
```