Programación Avanzada 2020

Laboratorio 1

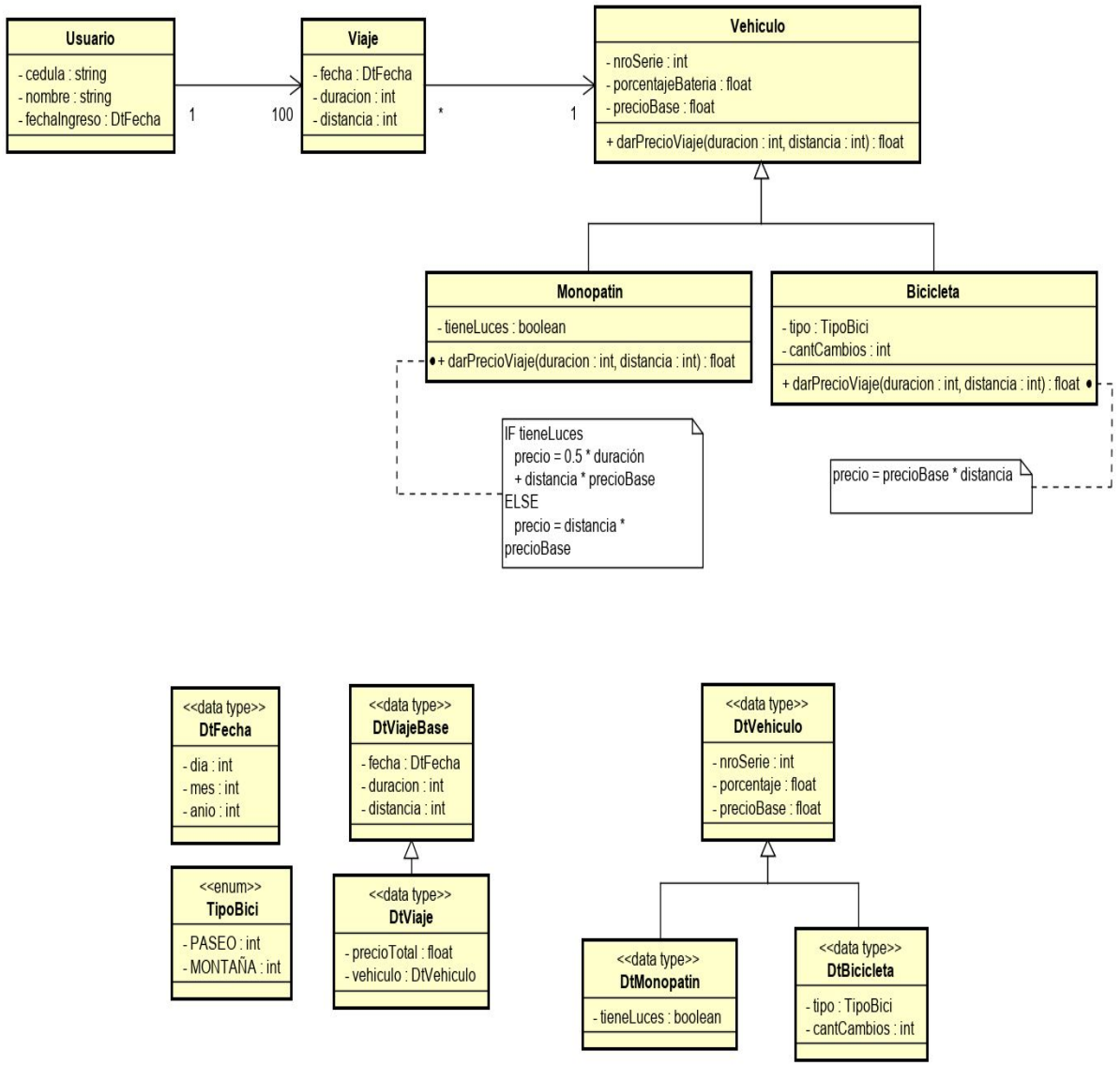
Consideraciones generales:

- La entrega podrá realizarse hasta la fecha indicada en el aula virtual de Programación Avanzada dentro del Campus**.**
- Las entregas deberán realizarse de acuerdo a las plantillas disponibles en el Campus.
- Las entregas serán realizadas **únicamente** vía web se deberán subir usando el Campus del curso. Sólo **un miembro** del grupo deberá entregar **un único archivo** que contenga la entrega, el archivo deberá llamarse **<número de grupo>_lab1.zip (o tar.gz)** que contenga:
  - El código fuente de los dos ejercicios.
  - Un archivo makefile, que permita compilar y ejecutar el código, independiente de cualquier entorno de desarrollo integrado (IDE).
  - Un archivo denominado resp_lab1.txt, con las respuestas a las preguntas planteadas en el ejercicio 2.
- Las entregas que no cumplan estos requerimientos no serán consideradas.
- El código junto con el makefile se probarán en alguna máquina con Linux en una defensa **obligatoria y eliminatoria con todos** los integrantes del grupo.
- **No se aceptarán entregas fuera del plazo establecido** y el hecho de no realizar una entrega implica la insuficiencia del laboratorio completo.

Con este laboratorio se espera que el estudiante adquiera competencias en la implementación de operaciones básicas, el uso básico del lenguaje C++ (que se usará en el laboratorio) y el entorno de programación en linux, así como reafirmar conceptos presentados en el curso. También se espera que el estudiante consulte el material disponible en el Campus del curso y que recurra a Internet con espíritu crítico, identificando y corroborando fuentes confiables de información.

Ejercicio 1

Se desea implementar un sistema que permita registrar los viajes que realizan los usuarios de vehículos eléctricos. Para tal fin, resulta de interés contar con información sobre los usuarios, los vehículos eléctricos que utilizan para viajar y los viajes que realizan. Se muestra a continuación un diagrama utilizado para representar dicha realidad.



De cada usuario se conoce su cédula (que lo identifica), su nombre y su fecha de ingreso al sistema. Un usuario puede realizar varios viajes, de los que se conoce el vehículo usado, la fecha en la que realizó el viaje, su duración y la distancia recorrida.

A su vez, de cada vehículo eléctrico se tiene su número de serie (que lo identifica), el porcentaje de carga de disponible de su batería y su precio base. Por ahora, un vehículo puede ser un monopatín o una bicicleta. De los monopatines se conoce si tienen luces, y de las bicicletas su tipo (puede ser de paseo o de montaña) y la cantidad de cambios.

Se quiere brindar la funcionalidad de que los usuarios consulten información de sus viajes. A

los efectos de la implementación de esa funcionalidad se decidió delegar a la representación de Vehículo, el cálculo del precio total de un viaje por medio de la operación darPrecioViaje. Esta operación devuelve el precio total según el tipo de vehículo, la distancia y la duración. La fórmulas según sea monopatín o bicicleta están definidas en el diagrama por notas conectadas con líneas punteadas.

Se pide:

Implementar en C++:

1. Todas las clases (incluyendo sus atributos, pseudoatributos, getters, setters, constructores y destructores), enumerados y datatypes que aparecen en el diagrama.
2. Una función main que implemente las siguientes operaciones:

    a. `void registrarUsuario(string ci, string nombre)`

    Registra un en el sistema. La fecha de ingreso se obtiene del reloj de la máquina. Si existe un usuario registrado con la misma cédula, se levanta una excepción `std::invalid_argument`.

    b. `void agregarVehiculo(DtVehiculo& vehiculo)`

    Agrega un nuevo vehículo al sistema. Controlar que se cumplen: (1) no existe un vehículo con el mismo número de serie, (2) porcentaje como valor entre 0 y 100 y (3) precio base positivo. De no ser así, se levanta una excepción `std::invalid_argument`.

    c. `void ingresarViaje(string ci, int nroSerieVehiculo, DtViajeBase& viaje)`

    Crea un viaje entre el usuario y el vehículo identificados por `ci` y `nroSerieVehiculo`, respectivamente. Controlar que se cumplen: (1) existe un usuario registrado con esa ci, (2) existe un vehículo registrado con ese nro serie, (3) duración y distancia positivas y (4) fecha del viaje posterior o igual a la fecha de ingreso del usuario. De no ser así, se levanta una excepción std::invalid_argument.

    d. `DtViaje** verViajesAntesDeFecha(DtFecha& fecha, string ci, int& cantViajes)`

    Devuelve un arreglo con información detallada de los viajes realizados por el usuario antes de cierta fecha. Para poder implementar esta operación se deberá sobrecargar el operador < (menor que) del tipo de datos (*data type*) *DtFecha*. `cantViajes` es un parámetro de salida donde se devuelve la cantidad de viajes encontrados (corresponde a la cantidad de valores *DtViaje* que se devuelven).

    e. `void eliminarViajes(string ci, DtFecha& fecha)`

    Elimina los viajes del usuario identificado por `ci`, realizados en la fecha ingresada. Si no existe un usuario registrado con esa cédula, se levanta una excepción `std::invalid_argument`.

    f. `void cambiarBateriaVehículo(int nroSerieVehiculo, float`

`cargaVehiculo)`

Modifica el porcentaje de carga de la batería del vehículo identificado por `nroSerieVehiculo`. En caso de que el vehículo no exista, o la carga ingresada no se encuentre entre 0 y 100 se levanta una excepción `std::invalid_argument`.

g. `DtVehiculo** obtenerVehículos(int& cantVehiculos)`

Devuelve un arreglo con los vehículos del sistema. `cantVehiculos` es un parámetro de salida donde se devuelve la cantidad de vehículos (corresponde a la cantidad de valores *DtVehiculo* que se devuelven).

3. Implementar en el main un menú sencillo que sea interactivo con el usuario para poder probar las funcionalidades requeridas en el punto 2. Al ejecutar el programa debe pedir el ingreso de un número especificando la acción a realizar, y pedir luego los datos necesarios para cada operación.

```
Ejemplo:
      Bienvenido. Elija la opción.
         1) Registrar usuario
         2) Agregar vehículo
         3) Ingresar viaje
            ….
         0)  Salir
         Opción:
```

4. Sobrecargar el operador de inserción de flujo (ej. `<<`) en un objeto de tipo `std::ostream`. Este operador debe "imprimir" las distintas clases de *DtVehiculo* (*DtMonopatin, DtBicicleta*) con el siguiente formato:

```
- Número serie:
- Porcentaje bateria:
- Precio base: $ XXX .
- (En caso de ser Monopatin):
      - Tiene luces: Si / No
- (En caso de ser Bicicleta):
      - Tipo de Bicicleta:
      - Cantidad de cambios:
      -
```

Consideraciones:

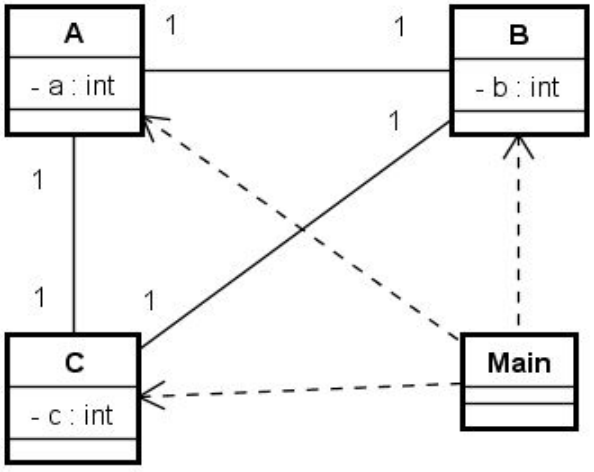
A los efectos de este laboratorio, la función main mantendrá una colección de usuarios, implementada como un arreglo de tamaño `MAX_USUARIOS`. De forma análoga sucede para los vehículos, con un arreglo de tamaño `MAX_VEHICULOS`. Puede implementar operaciones en las clases dadas en el modelo si considera que le facilitan para la resolución de las operaciones pedidas en el main.

Se puede utilizar el tipo `std::string` para implementar los atributos de tipo `string`.

El main debe manejar las excepciones lanzadas por las operaciones de los objetos.

Ejercicio 2

En este ejercicio se busca que se familiarice con la problemática de dependencias circulares en C++. Para esto se debe implementar y compilar la siguiente estructura dada por el diagrama a continuación.



Se pide:

1. Implementar, compilar y ejecutar en C++ el diagrama dado. Defina un atributo y constructores en las clases A, B y C así como operaciones que impriman un mensaje con el nombre de la clase y el valor del atributo. Defina un main que cree objetos de esas clases e invoque a dichas operaciones.
2. Responder las siguientes preguntas:
    a. ¿Cuáles son las dependencias circulares que fueron necesarias solucionar para que el programa compile?
    b. ¿Qué es forward declaration?