Programación 2

Examen de teoría (julio 2019)

4 de julio de 2019

Instrucciones

- **Duración: 3 horas**
- El fichero del primer problema debe llamarse `gestor.cc`. Para el segundo problema es necesario entregar cuatro ficheros, llamados `Tweet.cc`, `Tweet.h`, `User.cc` y `User.h`. Pon tu DNI y tu nombre en un comentario al principio de todos los ficheros fuente.
- La entrega se realizará como en las prácticas, a través del servidor del DLSI (`http://pracdlsi.dlsi.ua.es`), en el enlace `Programación 2`. Puedes realizar varias entregas, aunque sólo se corregirá la última.
- En la página web de la asignatura `http://www.dlsi.ua.es/asignaturas/p2` tienes disponibles algunos ficheros que te pueden servir de ayuda para resolver los problemas del examen, así como el apartado `Reference` de la web `www.cplusplus.com`.

Problemas

1. **(4.5 puntos)**

   Nuestra empresa ha cambiado de programa de gestión y necesitamos migrar los datos de los clientes al nuevo formato. El objetivo de este problema es desarrollar un programa que reciba como entrada los datos en el formato antiguo y los convierta al nuevo. El programa deberá ser capaz de realizar dos tareas: (i) leer el formato original en binario y transformarlo a un formato intermedio en un fichero de texto y, (ii) a partir de este fichero de texto, generar el formato destino en binario.

   Los datos del programa antiguo están almacenados en un fichero binario cuyos registros tienen la siguiente estructura:

   ```
   char name[20];          // Nombre del cliente
   char surname1[20];      // Primer apellido
   char surname2[20];      // Segundo apellido
   char address[100];      // Dirección postal
   int balance;            // Saldo en la cuenta de cliente
   ```

   El programa debe leer este fichero binario y guardar la información en un fichero de texto, donde la información de cada cliente ocupará una línea distinta con el siguiente formato:

   ```
   name|surname1|surname2|address|balance
   ```

   Si algún usuario del fichero binario tiene saldo (`balance`) negativo, en lugar de ese saldo negativo se almacenará en el fichero de texto el valor `NA` (*not available*, no disponible). Por ejemplo:

   ```
   Pablo|Castro|Ortega|C/ Mota del Carmen, 36, Sevilla|350
   Ana|Escobar|Soriano|C/ Alhambra, 22, Madrid|420
   Luis|Esteban|Torres|C/ San Vicente, 33, Alicante|NA
   ```

   A partir de este fichero de texto, el programa debe ser capaz de leer la información y guardarla en un fichero binario con el formato del programa nuevo. Los registros del nuevo programa tendrán esta estructura:

   ```
   char fullName[75];      // Nombre y apellidos
   char address[75];       // Dirección postal
   float balance;          // Saldo en la cuenta de cliente
   ```

En `fullName` se almacenarán el nombre y los dos apellidos, separados por un espacio en blanco. En `address` se almacenará la dirección, al igual que se tenía en el formato antiguo, pero teniendo en cuenta que el nuevo campo tiene un tamaño inferior al original y deberá recortarse el texto para que quepa. El campo `balance` contiene la misma información que en el original, aunque en este caso se trata de un campo de tipo `float` en lugar de `int`. Si el saldo leído del fichero de texto es `NA`, se le asignará el valor `0` en el nuevo formato.

El programa recibirá un parámetro de entrada para indicar qué tipo de operación queremos realizar con él:

- La opción `-l` permitirá leer un fichero binario en formato antiguo y generar un fichero de texto intermedio con el formato mencionado arriba
- La opción `-s` permitirá leer un fichero de texto intermedio y guardar los datos en un fichero binario con el nuevo formato

En ambos casos se recibirán dos parámetros más: primero el nombre del fichero binario y después el nombre del fichero de texto. Por ejemplo:

```
$ gestor -l datos_old.bin temp.txt
```

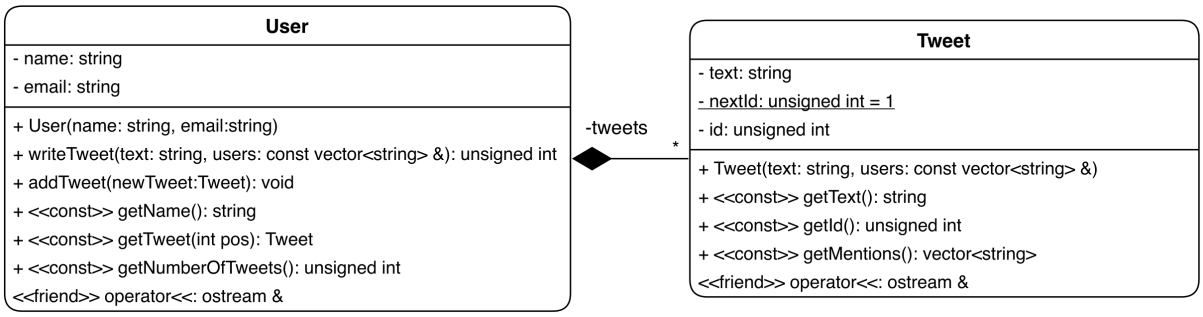
leerá los datos de clientes en formato antiguo de `datos_old.bin` y los guardará en el fichero de texto intermedio `temp.txt`, en el formato antes mencionado. Por otra parte:

```
$ gestor -s datos_new.bin temp.txt
```

leerá los datos del fichero intermedio `temp.txt` y generará un fichero binario `datos_new.bin` con el nuevo formato.

Los parámetros siempre se suministrarán en ese orden y no hará falta comprobarlo: primero la opción, seguida del nombre del fichero binario y seguida del nombre del fichero de texto. Lo que sí habrá que comprobar es que el número de parámetros es correcto y que el primero de ellos es `-l` o `-s`. En caso contrario se mostrará un error y se saldrá del programa.

2. **(5.5 puntos)**



Queremos hacer un programa para gestionar una versión simplificada de Twitter. Esta aplicación permite a un usuario escribir mensajes de texto, llamados *tweets* en la jerga de la aplicación. Como puedes ver en el diagrama, este programa tendrá dos clases: `User` (usuarios que envían mensajes) y `Tweet` (los propios mensajes).

Los objetos de la clase `Tweet` representan un mensaje o *tweet*. En el fichero `Tweet.h` deberás declarar el siguiente tipo enumerado, que contiene las posibles excepciones que se pueden producir en este programa:

```
enum Exception { exception_mention, exception_name, exception_email, exception_pos };
```

El texto de un *tweet* puede mencionar a uno o más usuarios utilizando el símbolo `@` antes de su nombre. El nombre de un usuario lo forman únicamente caracteres alfanuméricos (letras y números).[1] Por ejemplo, el *tweet* `are you sure it was me, @juan?` menciona al usuario de nombre `juan`.

[1] Puedes usar la función `isalnum` de la librería `cctype`, que recibe como único parámetro un carácter y devuelve `true` si es alfanumérico y `false` si no lo es.

El constructor de `Tweet` recibe, además del texto del *tweet*, un vector de nombres de usuarios que se utiliza para comprobar los usuarios mencionados en el texto de ese *tweet*: si se menciona en el texto a un usuario no incluido en el vector se lanzará una excepción de tipo `exception_mention`. Cada *tweet* tendrá un identificador numérico único, comenzando por 1. El identificador a usar para un nuevo *tweet* será el valor que tenga en ese momento el atributo estático `nextId`, que será incrementado en este constructor para poder ser usado en el siguiente *tweet*.

El método `getMentions` devuelve una lista con los nombres de los usuarios mencionados en el texto del *tweet*. Esta lista no deberá contener duplicados, de manera que si un usuario se menciona varias veces solo se incluirá una. Por último, el operador `<<` sobre un objeto de la clase `Tweet` muestra su identificador entre corchetes seguido de un espacio en blanco y del texto del *tweet*. Por ejemplo:

```
[3] just saw @arya at the supermarket!
```

Por lo que respecta a un usuario (`User`), éste tiene un nombre (`name`) y una dirección de correo electrónico (`email`). El constructor de la clase `User` recibe ambos valores. Primero deberá comprobar que el nombre del usuario es correcto, es decir, que sólo contiene caracteres alfanuméricos. En caso contrario, lanzará la excepción `exception_name`. En segundo lugar comprobará que la dirección de correo electrónico esté bien formada. En caso de que no lo esté, lanzará una excepción de tipo `exception_email`. Para simplificar, en este problema se asume que una dirección de correo electrónico está bien formada si empieza por una secuencia de uno o más caracteres alfanuméricos, seguidos del carácter `'@'`, y éste a su vez seguido de otra secuencia de caracteres alfanuméricos que debe contener al menos un punto (`'.'`), que no puede estar ni en la primera ni en la última posición de esa secuencia. Por ejemplo, `pepe@.com` o `pepe@com.` no serían válidos, porque el `'.'` se haya al comienzo de la secuencia en el primer caso y al final de la misma en el segundo.

El método `writeTweet` crea un objeto de la clase `Tweet` y lo añade a la lista de *tweets* del usuario. En caso de que el constructor de la clase `Tweet` lance una excepción, este método lo capturará y mostrará un error por pantalla indicando que no se ha podido crear el *tweet*. Si se ha creado correctamente, este método devuelve su identificador y en caso contrario devuelve `0`. El método `addTweet` permite añadir un *tweet* a la lista de mensajes enviados por el usuario. Asumiremos que el *tweet* pasado como parámetro se ha creado correctamente, por lo que no hay que hacer ninguna comprobación adicional en este método.

El método `getTweet` devuelve el *tweet* que ocupa la posición `pos` en la lista de *tweets* del usuario. Si la posición está fuera del rango del vector, lanzará una excepción de tipo `exception_pos`. El método `getNumberOfTweets` devuelve el número de *tweets* escritos por el usuario. Finalmente, el operador `<<` sobre un objeto de la clase `User` muestra los datos de un usuario y sus mensajes, siguiendo el formato del siguiente ejemplo:

```
juan (johnny5@google.com)
[1] just saw @arya at the supermarket!
[2] @juan and @arya bought the same cookies :-)
[3] good afternoon, world
```

Nota: No se pueden añadir nuevos métodos públicos al diseño proporcionado, pero sí privados.

Dado el siguiente fichero `main.cc`, que puedes compilar con `g++ Tweet.cc User.cc main.cc -o ej2`:

```
int main()
{
  vector<string> users;

  User juan("juan","john666@google.com");
  users.push_back(juan.getName());

  User arya("arya12","aryaPulova@hotmail.es");
  users.push_back(arya.getName());

  juan.writeTweet("just saw @arya12 at the supermarket!",users);
  arya.writeTweet("are you sure it was me, @juan?",users);
  Tweet t1("good morning, world",users);
  arya.addTweet(t1);
  Tweet t2("good afternon, world",users);
  juan.addTweet(t2);

  cout << juan << endl;
  cout << arya << endl;
}
```

La salida debería ser:

```
john (john666@google.com)
[1] just saw @arya12 at the supermarket!
[4] good afternoon, world

arya12 (aryaPulova@hotmail.es)
[2] are you sure it was me, @juan?
[3] good morning, world
```