Barker

La startup Barker S.L. nos ha contratado para llevar a cabo el desarrollo de una nueva red social llamada Barker.

El mecanismo de Barker es muy parecido al de otras redes sociales como Twitter, con una pequeña pero importante diferencia: mientras que otras plataformas prometen contenidos de calidad y argumentadas discursiones que nunca se cumplen, Barker es una red social honesta. A Barker se va a discutir porque sí, sin tener ni idea del tema de la discursión. Nuestra labor será implementar dicha red social desde cero, usando el lenguaje de programación C++.

Para ello, se adjunta una descripción de los requisitos y características del programa, así como de las distintas clases que lo forman. Además, nuestro equipo de QA ha desarrollado una serie de tests unitarios con las especificaciones requeridas, de forma que podemos testear en cualquier momento el código para comprobar nuestros avances. Pese a que es opcional usar los tests para el desarrollo del proyecto (es decir, podéis desarrollarlo ignorando los tests y escribiendo el código por vuestra cuenta), usaremos esos tests para corregir el código, además de mirar el código en si para valorarlo, por lo que se recomienda que los uséis.

La empresa nos ha pedido una entrega escalonada en varios hitos que se detallan al final de este documento. El trabajo se llevará a cabo en grupos de 2 (salvo que el total de alumnos no sea un número par, en cuyo caso habrá algún grupo de 1 alumno). Barker S.L. es una empresa de proyección internacional, por lo que el idioma usado para el nombre de las clases, métodos y atributos será el inglés.

Requisitos del programa

(es importante respetar la nomenclatura propuesta para clases, miembros y atributos para el correspondiente funcionamiento de los tests unitarios)
Las partes sombreadas en verde corresponden con elementos que, pese a pertenecer a una clase requerida en la primera entrega, no es necesario implementarlos hasta la segunda entrega.
Las partes sombreadas en azul corresponden con elementos que, pese a pertenecer a una clase requerida en la primera entrega, no es necesario implementarlos hasta la tercera entrega.

Descripción del funcionamiento general

El programa Barker está compuesto por tres componentes principales: usuarios, publicaciones y un manager que gestiona ambos. La aplicación estará construida alrededor del manager, y todas las acciones que realice el usuario en la plataforma a través de la interfaz del programa serán llevadas a cabo por el manager. El manager podrá registrar uno o varios usuarios, que podrán loguearse en el sistema para realizar editar sus perfiles, realizar publicaciones o seguir a otros usuarios y ver sus perfiles. Las publicaciones podrán ser de distintos tipos, todos ellos explicados en la sección correspondiente.

Usuarios

Cada usuario del sistema estará representado por una instancia de la clase *User*. Ya que cada usuario tiene cierta información que deber ser pública (p. ej. nombre de usuario y bio) y otra que debe de ser privada (p. ej. email y contraseña), existen dos clases: *PublicUserData* y *PrivateUserData*, que contienen esos datos. A continuación se describen los métodos y atributos de cada una de estas clases:

User

- El nombre de la clase es *User*
- Posee un constructor paramétrico que toma como argumentos el *email*, *password*, *username* (nombre de usuario) y la *bio* (breve descripción del usuario). Todos esos atributos son strings de C++ (tipo std::string).
- El constructor por defecto debe ser privado, ya que no se espera que se pueda crear un *User* sin especificar sus datos.
- El resto de sus miembros y atributos los hereda de las clases *PubicUserData* y *PrivateUserData* por herencia pública.

PrivateUserData

- El nombre de la clase es *PrivateUserData, y* representa los datos privados del usuario.
- Posee dos atributos, ambos de tipo std::string: el *email* y el *password* del usuario, ambos protegidos (protected).
- El constructor por defecto debe ser privado, ya que no se espera que se pueda crear un *PrivateUserData* sin especificar sus datos.
- El constructor paramétrico toma como argumentos el *email* y *password.* Este constructor será protegido (protected), para evitar su uso fuera de la cadena de herencia de la clase.
- Todos los atributos tienen sus correspondientes getters y setters, con formato *getAtributo()* y *setAtributo()* (importante respetar las mayúsculas).

PublicUserData

- El nombre de la clase es *PublicUserData, y repre*senta los datos públicos del usuario.
- Posee los siguientes atributos, todos ellos protegidos (protected).
  - *username* (nombre de usuario) y *bio* (breve descripción del usuario), ambos de tipo std::string y con sus correspondientes getters y setters.
  - *followers,* de tipo entero, que almacena el número de usuarios que siguen a este usuario. Este atributo tiene un getter *getFollowers(),* pero en lugar de un setter dispone de dos miembros *increaseFollowers()* y *decreaseFollowers()* que incrementan y decrementan el número de followers en una unidad cada vez que son llamados. El número de followers nunca puede ser menor de 0.
  - *following,* un std::vector de punteros a *PublicUserData,* que apuntan a los usuarios a los que el usuario actual sigue. Este miembro posee un getter *getFollowing(),* pero no un setter.
  - *publications,* un std::vector de punteros a *Publication,* que contiene todas las publicaciones que ha realizado el usuario actual. Este miembro posee un getter *getPublications(),* pero no un setter.
- Además, posee los siguientes métodos:
  - Un constructor por defecto privado, ya que no se espera que se pueda crear un *PublicUserData* sin especificar sus datos.

- Un constructor paramétrico que toma como argumentos el *username* y la *bio*, ambos de tipo std::string. No es necesario especificar ningún otro argumento explícitamente, el resto se inicializarán a sus valores por defecto (por ejemplo, el número de followers será inicialmente 0 en todos los casos).
- Para añadir y eliminar seguidores, se dispone de los siguientes métodos:
  - *follow*(), que toma como argumento un puntero al usuario al que seguir, y lo incorpora a la lista de usuarios a los que sigue el usuario actual. Devuelve true en caso de éxito y false en caso contrario.
  - *unfollow*(), que toma como argumento un puntero al usuario al que dejar de seguir, y lo elimina de la lista de usuarios a los que sigue el usuario actual. Devuelve true en caso de éxito y false en caso contrario.
- Para añadir y eliminar publicaciones, se dispone de los siguientes métodos (las publicaciones no se pueden editar):
  - *addPublication*(), que toma un puntero a la publicación que se desea incorporar a a lista de publicaciones del usuario. Devuelve true en caso de éxito y false en caso contrario.
  - *removePublication*(), que toma un entero que representa el id de la publicación a eliminar de la lista de publicaciones del usuario. Devuelve true en caso de éxito y false en caso contrario.

Manager

La clase *Manager* es la encargada del funcionamiento de la aplicación, y de proporcionar toda la funcionalidad de la misma.

Para ello cuenta con los siguientes atributos:

- *users,* un std::vector de punteros a *User,* que almacena a los usuarios actualmente registrados en el sistema.
- *currentUser,* un entero que indica qué usuario es el que está actualmente logueado en el sistema, indicando su posición en el vector *users*. Cuando todavía no hay ningún usuario logueado, este valor debe ser -1.
- Ambos miembros no poseen los típicos getters y setters, ya que el acceso a estos miembros debe estar controlado por el sistema. En la sección dedicada a los miembros de esta clase se explica con detalle qué métodos permiten el acceso a estas variables y cómo.

Además de los anteriores atributos, debe poseer los siguientes métodos generales:

- Un constructor por defecto que inicializa los atributos de la clase.
- Un destructor que se encarga de eliminar todos los objetos creados dinámicamente por el *manager.*

Cuando no hay ningún usuario logueado, el *manager* debe poder realizar las siguientes acciones:

- Crear un nuevo usuario a través del método *createUser(),* que toma como argumentos el *email*, *password*, *username* y la *bio*. Todos ellos son de tipo std::string. Este método crea un usuario nuevo y lo almacena en la lista de usuarios registrados. Devuelve true en caso de éxito y false en caso contrario.
- Mostrar todos los usuarios registrados a través del método *showUsers().* Para evitar proporcionar al usuario datos privados de otros usuarios, este método devuelve un vector de punteros a *PublicUserData,* exponiendo sólo la información pública de los usuarios registrados.
- Mostrar un único usuario registrado a través del método showUser(), que toma una std::string con el *username* del usuario que se desea mostrar y devuelve su información

pública en forma de puntero a *PublicUserData*. En caso de que el usuario pedido no exista, esta función devuelve un *nullptr*.

- Loguearse en el sistema a través del método *login()*, que toma como argumentos el *email* y *password* de un usuario, ambos de tipo std::string y los valida. Un usuario que no existe no se puede loguear en el sistema, un usuario ya logueado no se puede loguear de nuevo, y no se puede loguear un usuario con una contraseña incorrecta. En todos esos casos debe devolver false, y en caso de éxito, true.

Cuando un usuario se encuentra logueado, puede hacer uso de las siguientes acciones (en caso contrario estos métodos deben devolver false):

- Conocer si hay un usuario logueado a través del método *isLogged()*, que devuelve true en caso de que haya un usuario logueado, y false en caso contrario.
- Cerrar sesión a través del método *logout()*, que hace que el usuario actual deje de estar logueado. Devuelve true en caso de éxito y false en caso contrario. Intentar cerrar sesión sin que haya un usuario logueado se considera un comportamiento erróneo.
- Obtener toda la información (pública y privada) sobre el usuario logueado a través del método *getCurrentUser()*, que devuelve un puntero al *User* que se encuentra actualmente logueado. En caso de no haber un usuario logueado, esta función devuelve *nullptr*.
- Dar de baja (eliminar) al usuario logueado actualmente a través del método *eraseCurrentUser()*. Devuelve true en caso de éxito y false en caso contrario.
- Editar la información del usuario logueado a través de las funciones *editEmail()*, *editPassword()*, *editUsername()* y *editBio()*, que toman una std::string y modifican el atributo correspondiente del usuario actual. Estos métodos devuelven true en caso de éxito y false en caso contrario.
- Seguir o dejar de seguir a un usuario a través de los métodos *followUser()* y *unfollowUser()*, los cuales toman como argumento una std::string con el *username* del usuario al que queremos seguir o dejar de seguir. Estos métodos devuelven true en caso de éxito y false en caso contrario.

Una vez tengamos que desarrollar el código para las publicaciones (segunda entrega), deberemos añadir los siguientes métodos:

- El conjunto de publicaciones de un usuario se conoce en Barker como Feed. Con el método *GetUserFeed()* podemos obtener un std::vector de punteros a *Publication* con todas las publicaciones que ha hecho un usuario. El usuario se especifica mediante un argumento de tipo std::string que es pasado a este método. No es necesario estar logueado para usar este método.
- El conjunto de publicaciones de todos los usuarios a los que sigues se conoce en Barker como Timeline. Con el método *getTimeline()* podemos obtener un std::vector de punteros a *Publication* con todas las publicaciones de los usuarios a los que el usuario actual sigue. Para usar esta función es necesario estar logueado, en caso contrario devuelve un std::vector vacío.
- Para crear publicaciones se usarán los métodos *createBark()*, *createRebark()* y *createReply()* según el tipo de publicación a crear (los tipos de publicación se encuentran explicados en la sección correspondiente), que será almacenada en el usuario actualmente logueado. El método *createBark()* toma como argumento el texto del Bark a ser creado en forma de std::string, mientas que *createRebark()* y *createReply()* toman un entero que representa el id de la publicación a la que hacen referencia, además del texto en forma de std::string. Todos ellos devuelven true en caso de éxito y false en caso contrario.

Además de las funciones mencionadas anteriormente, se deberá poder cargar y guardar el estado del sistema a través de los siguentes métodos, usando ficheros (requisito de la tercera entrega):

- *saveToFile()*, que toma como argumento la ruta del archivo a guardar en forma de std::string, y vuelca toda la información del sistema al fichero (*User*s y *Publication*s).
- *loadFromFile()*, que toma como argumento la ruta del archivo a cargar en forma de std::string, y restaura toda la información guardada en el fichero (*User*s y *Publication*s).

El formato de los ficheros se detalla al final de este documento.

Publicaciones

Las publicaciones son la forma que tienen los usuarios de expresarse (iniciar y continuar discursiones) en Barker. Hay tres tipos de publicación en Barker: Bark, Rebark y Reply (respuesta).

Publication

Publication es una clase abstracta que contiene todos los elementos comunes a los distintos tipos de publicaciones, además de ofrecer un método polimórfico *getBark()* para obtener una representación textual de cualquier publicación.

Para ello cuenta con los siguientes atributos:

- *id*, un entero con un valor único para cada publicación, que sirve para identificarla.
- *time*, un unsigned long int que representa cuándo se creó la publicación.
- *user*, un puntero a *PublicUserData* que representa al usuario que creó la publicación.

Todos ellos cuentan con sus correspondientes setters y getters.

El constructor de esta clase estará protegido (protected) y tomará el *id*, *time* y *user* como argumentos.

Además, cuenta con el método virtual puro *getBark()*, que cuya función es devolver el contenido de la publicación en forma de texto (std::string) con un formato determinado, dependiendo del tipo de publicación. El formato correspondiente a cada tipo de publicación se especificará en la sección correspondiente a dicho tipo.

Bark

Un *Bark* es la publicación estándar en Barker, un mensaje de texto que el usuario publica y que sus seguidores pueden compartir o responder para iniciar una discursión. Un Bark es un tipo de *Publication* que, además, cuenta con los siguientes atributos propios:

- *text*, una std::string que contiene el texto del Bark, con sus correspondientes getters y setters.

El constructor de la clase *Bark* toma como argumentos el *id*, *time*, *user* (todos ellos explicados en *Publication*), y una std::string con el texto.

El método *getBark()* deberá devolver una std::string con el contenido del *Bark* formateado correctamente. El formato esperado para un *Bark* es el siguiente: **"{username} - {time}:\n{text}"**, donde los elementos entre corchetes serán sustituidos por el valor correspondiente.

Rebark

Otro tipo de publicación posible es compartir un *Bark* que originalmente haya escrito otro usuario a nuestros seguidores. Esto en Barker es un *Rebark*, y es un tipo de *Publication.*

Un *Rebark*, además de los atributos heredados de *Publication*, cuenta con los siguientes propios:

- *publication*, un puntero a la *Publication* que estamos compartiendo con nuestros seguidores, con sus correspondientes getters y setters.
- *text*, una std::string que contiene un texto con algún comentario sobre la *Publication* original, con sus correspondientes getters y setters.

El constructor de Rebark toma como argumentos el *id*, *time* (explicados en *Publication*), *publication* (un puntero a la *Publication* a compartir), *user* (puntero al *PublicUserData* que hace el *Rebark*) y *text*, una std::string que contiene el texto del *Rebark*.

El método *getBark()* deberá devolver una std::string con el contenido del *Rebark* formateado correctamente. El formato esperado para un *Rebark* es el siguiente: **"{username} rebarked - {time}:\n{text}\n***\n{original bark}\n***"**, donde los elementos entre corchetes serán sustituidos por el valor correspondiente. **{original bark}** corresponde al texto que devuelve el método *getBark()* de la publicación original que se referencia.

Reply

Además de publicar o compartir mensajes, se puede responder a un mensaje existente. Esto en Barker es una *Reply* (respuesta), y es también un tipo de *Publication.*

Una *Reply*, además de los atributos heredados de *Publication*, cuenta con los siguientes propios:

- *publication*, un puntero a la *Publication* a la que se responde, con sus correspondientes getters y setters.
- *text*, una std::string que contiene el texto de la respuesta a la *Publication* original, con sus correspondientes getters y setters.

El constructor de Reply toma como argumentos el *id*, *time* (explicados en *Publication*), *publication* (un puntero a la *Publication* a la que se responde), *user* (puntero al *PublicUserData* que realiza la Reply) y *text*, una std::string que contiene el texto la respuesta.

El método *getBark()* deberá devolver una std::string con el contenido de la *Reply* formateado correctamente. El formato esperado para una *Reply* es el siguiente: **"{username} replied - {time}:\n===\n{original bark}\n===\"\n{text}"**, donde los elementos entre corchetes serán sustituidos por el valor correspondiente. **{original bark}** corresponde al texto que devuelve el método *getBark()* de la publicación original a la que se responde.

Interfaz del programa

Para poder interactuar con el usuario, el programa debe disponer de una interfaz que muestre los datos que devuelve el *Manager* al usuario, y que pida datos al usuario para introducirlos en el *Manager.* Esta interfaz se llevará a cabo a través de una terminal, usando las funciones de entrada y salida estándar de C++.

Esta interfaz nos deberá permitir llevar a cabo todas las acciones posibles que permite el *Manager* (crear un usuario, iniciar sesión, ver el Timeline / Feed, etc), controlando el tipo de datos que el usuario nos proporciona y qué funciones están expuestas según el usuario esté o no logueado.

El formato de la interfaz queda a criterio del alumno, valorándose la claridad e intuitividad de la misma. La interfaz será probada a mano durante la entrega final, por lo que no se proporcionan tests unitarios para esta parte.

Posibles mejoras

Además de la funcionalidad básica (necesaria para aprobar), la interfaz en terminal y el soporte para ficheros, el alumno puede implementar alguna de las siguientes mejoras (u otras propuestas por el alumno), que serán probadas a mano durante la entrega final.

- El argumento *time* de las publicaciones es tomado del reloj del sistema a través de las funciones de la librería ctime.
- Mostrar el Timeline del usuario ordenado cronológicamente.
- Pedir confirmación antes de borrar una cuenta de usuario.
- Si un *Bark* es eliminado, se eliminan los *Rebarks* (o todas las publicaciones) que hagan referencia a dicho *Bark*.
- Añadir una sobrecarga de operador, por ejemplo, para usar los operadores >> y << con streams de entrada / salida.

Funcionalidad a implementar en cada entrega

Primera entrega

La primera entrega estará compuesta por:

- Un diagrama UML del sistema a implementar.
- El código relativo a las clases de usuarios (test_barker_user).
- El código relativo a la funcionalidad básica del manager (test_barker_manager_basic).

Esta entrega no incluye el código relativo a las publicaciones, ni el soporte de ficheros, ni la interfaz por terminal, sólo se requiere que el código pase los tests mencionados anteriormente.

Fecha de entrega: 27 / 04 / 2020

Segunda entrega

La segunda entrega y última entrega estará compuesta por:

- El código relativo a las publicaciones (test_barker_publication).
- El código necesario para integrar las publicaciones con el código anterior (test_barker_publication_integration).

Esta es la funcionalidad básica que hay que implementar para poder aprobar el trabajo. Si todos los tests hasta este punto pasan con éxito, la nota del trabajo será un 5 (condicionada a los resultados de las cuestiones individuales a cada miembro durante la entrega final).

Además, se deberá entregar el código completo, incluyendo:

- Funcionalidad para guardar y cargar archivos (test_barker_load_save_to_file) (2pts)
- La interfaz del programa (3pts).
- Aquellas mejoras que cada grupo considere oportunas. (hasta 1pt por mejora, hasta 2pts en total).

Esta es la funcionalidad que da puntos por encima del aprobado (5 sobre 10), siendo el aporte de cada parte el especificado junto al requisito.

Tras la entrega se llevará a cabo una sesión online especial, en la que se preguntará a cada miembro del grupo de forma individual cuestiones sobre la implementación del código, de forma que podamos comprobar que ambos habéis contribuido de manera similar al trabajo.

Fecha de la entrega: 15 / 05 / 2020
(cada grupo será citado en un slot de tiempo, para que no tengáis que esperar mientras vuestros compañeros son evaluados).

Tests Unitarios

Consejos a la hora de testear el código:

1. Para que los tests compilen y funcionen, los archivos, clases y métodos se tienen que llamar exactamente igual a como aparecen en este documento, ya que si no cmake / gtest no los encontrará. El primer paso será crear los archivos necesarios, vacíos, para que cmake genere el proyecto correctamente.
2. Para poder compilar los tests y ejecutarlos, las clases y métodos que se usan en el test deben existir y estar implementados. Mi consejo es que lo primero que hagáis sea crear las clases y métodos vacíos, sin implementación alguna, hasta que estén todos los necesarios y los tests compilen. Si alguna función tiene que devolver un valor, haced que devuelva un valor cualquiera (el test compilará pero fallará, que es lo que queremos en este punto).
3. Una vez el test se pueda ejecutar y haya fallado, podemos comenzar a escribir el código dentro de las funciones que hemos creado en el paso anterior, hasta que los tests pasen con éxito.
4. Podéis (y deberíais) desactivar tests comentando el bloque correspondiente en el archivo CMakeLists.txt dentro de la carpeta tests. De esta forma sólo tenéis que centraros en el código de la parte que estáis desarrollando en ese momento.
5. De igual manera, sólo los tests de la primera entrega están activados por defecto. Para activar el resto debéis descomentar el bloque correspondiente en el archivo CMakeLists.txt dentro de la carpeta tests.
6. Los getters y setters de las clases tiene el formato *getAtributo()* y *setAtributo()*. Por ejemplo, para un atributo llamado *name* serán *getName()* y *setName()*,

Formato de los archivos de salida

- Al principio del archivo se volcarán los *User*s y, posteriormente, las *Publication.*
- El formato de cada usuario es el siguiente:
  - Un # al comienzo del bloque del usuario
  - email
  - password
  - username
  - bio
  - Número de seguidores
  - La string “following:”, que indica el comienzo de la enumeración de seguidores.
  - El username de los seguidores, un username por línea

- La string “publications:”, que indica el comienzo de la enumeración de publicaciones.
- El id de las publicaciones del usuario, un id por línea.
- Un # al final del bloque del usuario

Ejemplo:

```
#
fakeemail@domain.com
user3pass
user3
Third user of Barker
0
following:
coolGuy98
user2
publications:
3
4
#
```

- El formato de cada publicación es el siguiente:
  - Las palabras clave “$Bark”, “$Rebark” o “$Reply”, dependiendo del tipo de publicación, al inicio de cada bloque.
  - id de la publicación
  - time de la publicación
  - En el caso de un Rebark o Reply, el id de la publicación a la que hacen referencia
  - username del usuario que realiza la publicación
  - texto de la publicación, en una sola línea

Ejemplo:

```
$Bark
0
0
coolGuy98
This is my first bark. Hello, world!
$Rebark
1
20
0
user2
This guy is hilarious
```