

# Práctica 4:

## Gestor de correo *fdimail*

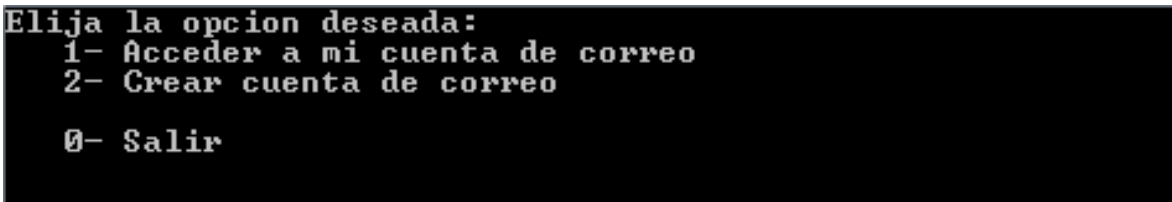
Fecha de entrega: 17 de mayo de 2015

### 1. Funcionalidad de la aplicación a desarrollar

Una *startup* española quiere desarrollar un gestor de correo electrónico que pueda rivalizar con los que se usan actualmente en el mercado, y ha decidido contar con los alumnos de Fundamentos de Programación de la UCM para que lo implementen. El prototipo será un gestor de correo local en modo consola, que permite a nuevos usuarios registrarse, y a los usuarios ya registrados enviar y recibir correos de manera sencilla.

#### 1.1. Registro e inicio de sesión

La pantalla principal del sistema (ver Figura 1) permite realizar dos opciones: acceder a una cuenta de correo ya existente o crear una cuenta de correo nueva. En ambos casos se pide un nombre de usuario y contraseña.



```
Elija la opcion deseada:  
1- Acceder a mi cuenta de correo  
2- Crear cuenta de correo  
  
0- Salir
```

Figura 1. Pantalla de inicio

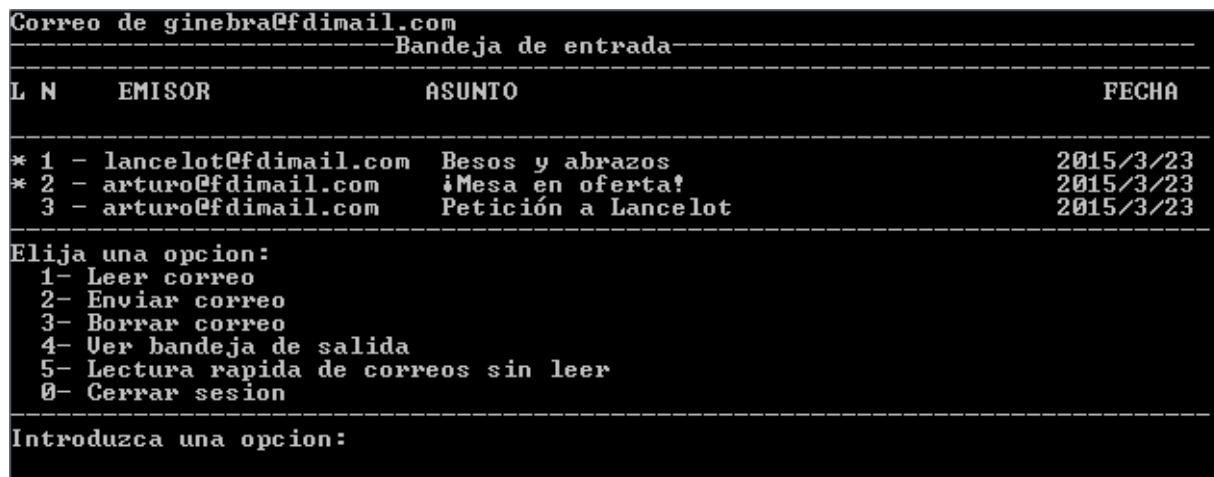
Cuando se elige la opción de **acceder a una cuenta de correo ya existente**, se comprueba que el usuario exista en la base de usuarios del sistema y que la contraseña sea correcta. Si una de estas dos condiciones no se cumple, se muestra un error y el menú principal aparece de nuevo.

Cuando se elige la opción de **crear una nueva cuenta**, se comprueba si el usuario que se quiere registrar ya existe en la lista de usuarios del sistema. Si el nuevo usuario ya existe se mostrará un error y se volverá al menú principal.

Si no ha habido error, y el usuario ha podido acceder a su cuenta o crear una cuenta nueva, se muestra la pantalla principal del gestor de correo.

## 1.2. Pantalla principal del gestor de correo

La pantalla principal del gestor de correo muestra la bandeja de entrada correspondiente al usuario registrado en la aplicación (ver Figura 2).



Correo de ginebra@fdimail.com

-----Bandeja de entrada-----

L	N	EMISOR	ASUNTO	FECHA
*	1	- lancelet@fdimail.com	Besos y abrazos	2015/3/23
*	2	- arturo@fdimail.com	¡Mesa en oferta!	2015/3/23
	3	- arturo@fdimail.com	Petición a Lancelot	2015/3/23

Elija una opcion:

- 1- Leer correo
- 2- Enviar correo
- 3- Borrar correo
- 4- Ver bandeja de salida
- 5- Lectura rapida de correos sin leer
- 0- Cerrar sesion

-----

Introduzca una opcion:

Figura 2. Pantalla principal del gestor de correo

La lista de correos recibidos se muestra numerada y en orden inverso de llegada (de más a menos reciente). Los correos no leídos se marcan con un asterisco. Se dispone además de un menú que permite realizar las siguientes operaciones:

- **Leer correo:** Solicita al usuario el número del correo a mostrar, comprueba dicho número, y si no ha habido error, muestra en una nueva pantalla el correo junto con las opciones que se pueden realizar sobre él (ver sección 1.3).
- **Enviar correo:** Permite al usuario escribir y enviar un nuevo correo. Se solicitarán los datos del nuevo correo en el siguiente orden:
  - El destinatario (sólo uno).
  - El asunto del correo (una línea).
  - El cuerpo del correo (varias líneas).

El resto de datos de un correo (identificador, emisor y fecha) se calcularán automáticamente (ver detalles de implementación).

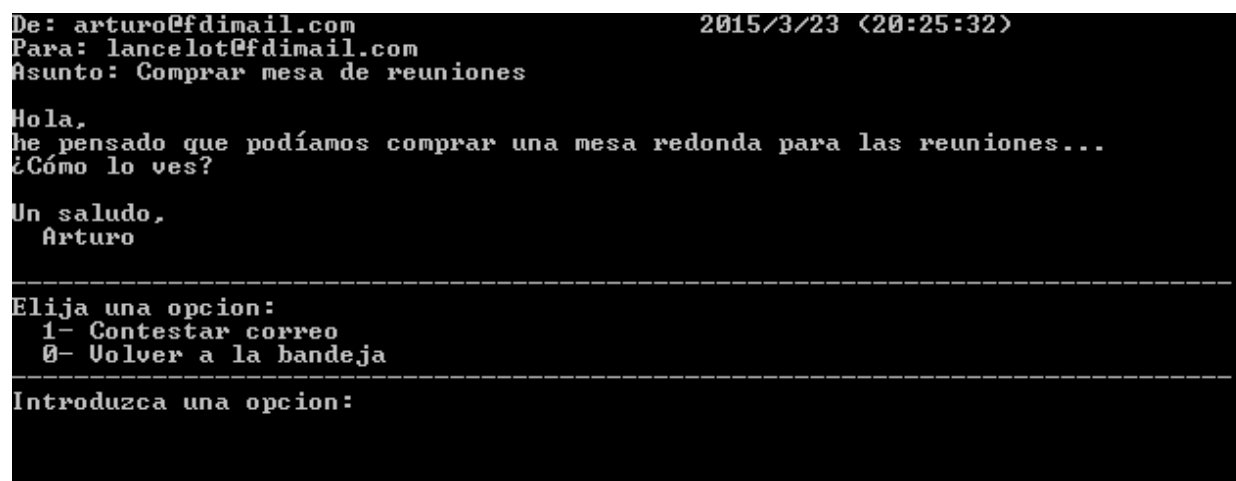
Si no se ha podido enviar el correo al destinatario (porque no existe o porque sus listas están llenas), se indicará mediante un mensaje de error que el envío a dicho destinatario no se ha podido realizar.

Si un correo se ha enviado con éxito al destinatario, dicho envío quedará reflejado en dos sitios: la bandeja de salida del emisor y la bandeja de entrada del destinatario.

- **Borrar correo:** Solicita un número de correo al usuario y lo borra de la bandeja de correos que se está mostrando.
- **Lectura rápida de correos sin leer:** Mostrará todos los correos sin leer de la bandeja correspondiente ordenados en primer lugar por asunto y después por fecha. **OJO:** Para ordenar por asunto debes ignorar los “Re: ” de los correos que sean una respuesta.
- **Ver bandeja de salida:** Cambia la vista para mostrar la bandeja de salida. La bandeja de salida se muestra exactamente igual que la de entrada, con dos excepciones: los correos que muestra son los enviados por el usuario y en lugar de mostrar la opción “Ver bandeja de salida” mostrará la opción “Ver bandeja de entrada”, que nos permite volver a la bandeja de entrada.

### 1.3. Pantalla de lectura de correo

Cuando el usuario elige la opción **Leer correo** el correo elegido se muestra en una nueva pantalla (ver Figura 3):



```
De: arturo@fdimail.com                2015/3/23 <20:25:32>
Para: lancelet@fdimail.com
Asunto: Comprar mesa de reuniones

Hola,
he pensado que podíamos comprar una mesa redonda para las reuniones...
¿Cómo lo ves?

Un saludo,
Arturo

-----
Elija una opcion:
1- Contestar correo
0- Volver a la bandeja
-----
Introduzca una opcion:
```

Figura 3. Pantalla donde se muestra el correo y las opciones del correo

Además se muestra un menú que incluye las siguientes opciones:

- **Contestar correo:** Generará un nuevo correo como contestación del correo original que se está mostrando. Los datos del nuevo correo contestación se rellenarán de la siguiente manera:
  - El emisor será el usuario que está actualmente trabajando en la aplicación.
  - El receptor será el emisor del correo original. No se permite añadir nuevos destinatarios a la contestación de un correo.

- El asunto será el mismo, pero se le añadirá al principio la cadena "Re: ".
- El cuerpo del correo se pedirá al usuario (varias líneas), y a lo escrito por el usuario se le añadirá al final todo el contenido del correo original (incluyendo los datos de emisor, destinatarios, fecha de envío y asunto).
- El resto de datos de un correo (identificador y fecha) se calcularán automáticamente (ver detalles de implementación).

Cuando se contesta a un correo, éste se envía al destinatario como un correo normal, y se guarda una copia del mismo en la bandeja de salida del usuario que lo ha contestado. Igual que al enviar un correo nuevo, si no se puede realizar la entrega se mostrará un mensaje avisando al usuario.

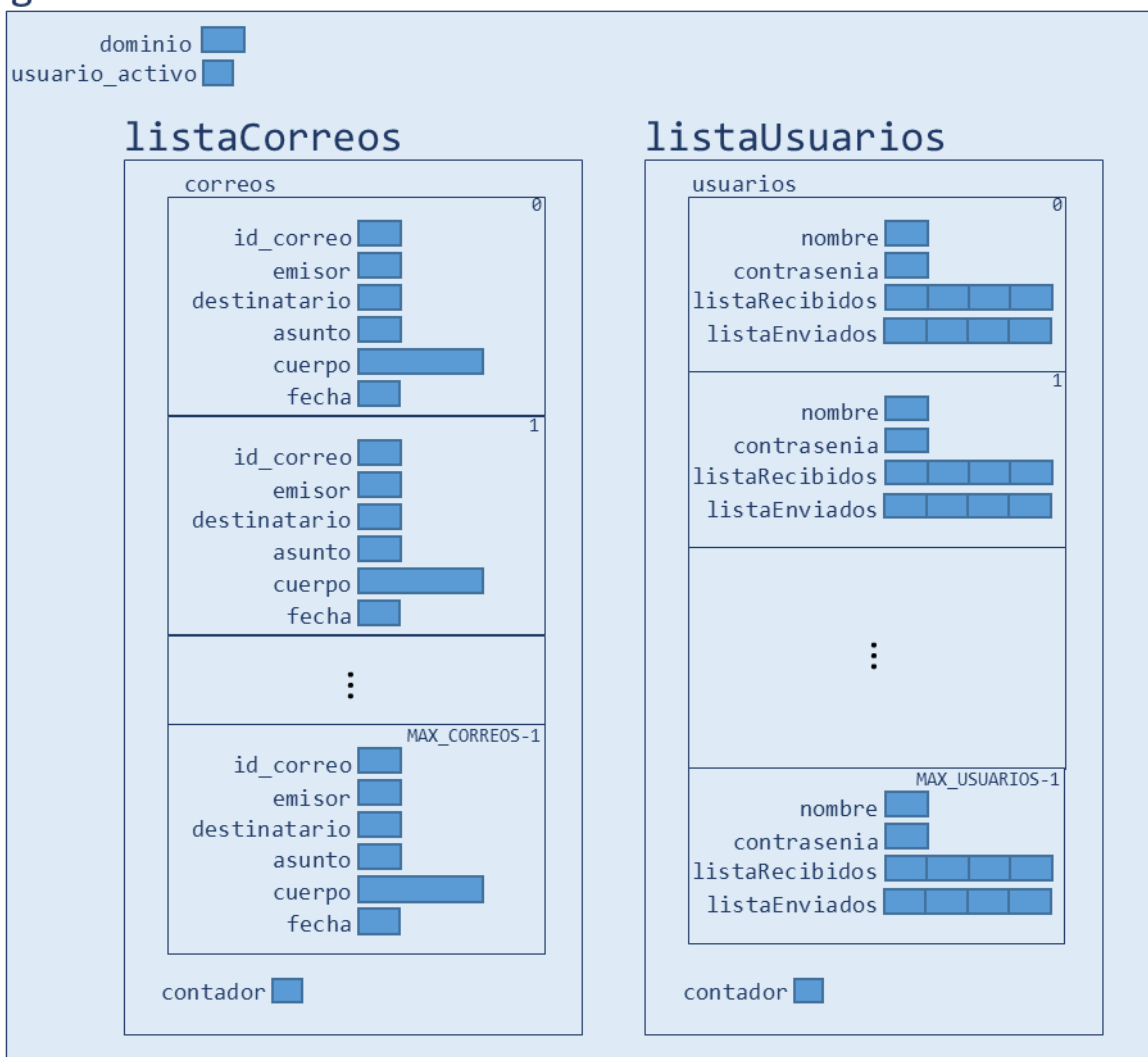
- **Volver a la bandeja:** Se volverá a la bandeja de correos (de entrada o salida) en la que se estaba antes de ejecutar la opción de leer un correo.

## 2. Visión general de la implementación: módulos de la aplicación y relación entre ellos

Hasta ahora hemos hablado de los requisitos de funcionamiento de la aplicación. Ahora vamos a explicar los requisitos de implementación.

La aplicación requiere implementar un tipo de datos `tGestor` que representará el gestor de correos y que implementará todas las funcionalidades de nuestra aplicación. El tipo de datos `tGestor` tendrá, entre otros campos, la lista de correos (tipo `tListaCorreos`) y la lista de usuarios (`tListaUsuarios`). Como es natural, esto supone que tenemos que implementar también los tipos `tCorreo` y `tUsuario`. La Figura 4 muestra una variable gestor de tipo `tGestor` para hacernos una idea.

## gestor

Figura 4. La variable gestor de tipo `tGestor`

Es importante tener en cuenta que **EL GESTOR SÓLO ALMACENA UNA COPIA DE CADA CORREO**, más concretamente, la almacena en la lista de correos. Por su parte, un usuario no almacena sus correos recibidos y enviados, sino que tanto su bandeja de entrada como su bandeja de salida serán una lista de registros que contendrán, entre otras cosas, un identificador que permitirá localizar el correo en la lista de correos. La Figura 5 muestra una representación de los campos `listaRecibidos` y `listaEnviados` que tendrá el usuario, ambos campos serán de tipo `tListaRegistros` y requerirán definir un tipo de datos `tRegistro`.

Vamos a verlo con un ejemplo en el que el usuario “arturo” ha enviado un correo al usuario “lancelot”. La Figura 6 representa la información relevante del estado del gestor en este ejemplo. El identificador de correo “arturo@fdimail.ucm.es\_314159” aparece en la lista de enviados de “arturo” y en la de recibidos de “lancelot”. Sin embargo, el correo propiamente dicho sólo se encuentra una vez en la lista de correos del gestor.

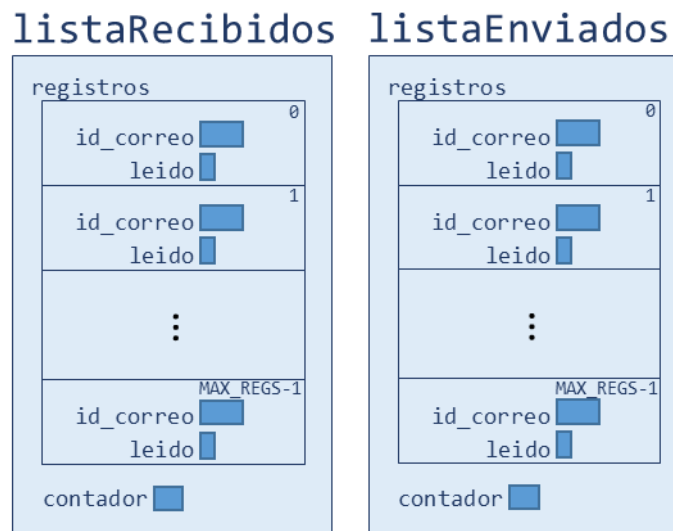


Figura 5. Los campos `listaRecibidos` y `listaEnviados` de tipo `tListaRegistros`



Figura 6. Ejemplo del estado del gestor en un determinado momento

## 2.1. Organización en módulos

La práctica deberá organizarse en módulos, con un módulo independiente para cada tipo principal. Se deben definir al menos los siguientes módulos:

### Módulo Correo

Declara un tipo de datos `tCorreo` para guardar la información de un correo:

- El emisor y el destinatario (`string`). Corresponderán con identificadores de usuarios en el sistema.
- El asunto y el cuerpo del correo (`string`).
- La fecha (`tFecha`). Ver más adelante.
- Un identificador **único** (`string`). Para que el identificador sea único, se compone de la concatenación del emisor y la fecha (p.e. `pepe@fdimail.com_143456443`).

Implementa al menos los siguientes subprogramas:

- `void correoNuevo(tCorreo &correo, string emisor)`: Recibe un identificador de emisor y devuelve un correo con todos sus datos rellenos según se ha explicado en la sección 1.3.
- `void correoContestacion(const tCorreo &correoOriginal, tCorreo &correo, string emisor)`: Recibe un identificador de emisor y el correo original que se va a contestar, y devuelve un correo con todos sus datos rellenos según se ha explicado en la sección 1.3.
- `string aCadena(const tCorreo &correo)`: Devuelve un `string` con el contenido completo del correo para poderlo mostrar por consola.
- `string obtenerCabecera(const tCorreo &correo)`: Devuelve un `string` que contiene la información que se mostrará en la bandeja de entrada/salida: emisor, asunto y fecha sin hora.
- `bool cargar(tCorreo &correo, ifstream& archivo)`: Dado un flujo de archivo de entrada (ya abierto), lee los datos que corresponden a un correo y lo devuelve. Devuelve `false` sólo si el correo cargado no es válido.
- `void guardar(const tCorreo &correo, ofstream& archivo)`: Dado un flujo de archivo de salida (ya abierto), escribe en el flujo los datos que corresponden a un correo.

## Módulo ListaCorreos

Declara un tipo de datos `tListaCorreos` para gestionar una lista de correos. Es **IMPORTANTE** que tengas en cuenta que esta lista se encuentra **ordenada por el identificador de correo**. Debes implementarla como una lista de tamaño variable.

Implementa al menos subprogramas para:

- `void inicializar(tListaCorreos &correos)`: Inicializa la lista.
- `bool cargar(tListaCorreos &correos, string dominio)`: Implementa la carga de la lista de correos desde el fichero de correos de nombre `<NombreDominio>_correos.txt`.
- `void guardar(const tListaCorreos &correos, string dominio)`: Implementa el guardado de la lista de correos en el fichero de correos de nombre `<NombreDominio>_correos.txt`. Guarda en fichero la lista de correos. El nombre del fichero será como en el subprograma anterior.
- `bool insertar(tListaCorreos &correos, const tCorreo &correo)`: Dado un correo, si hay espacio en la lista, lo coloca en la posición de la lista que le corresponda de acuerdo con su identificador y devuelve `true`. Si no lo ha podido colocar devuelve `false`.
- `bool buscar(const tListaCorreos &correos, string id, int &pos)`: Dado un identificador de correo y la lista, devuelve, si dicho identificador existe en la lista, su posición y el valor `true`, y si no existe en la lista, la posición que le correspondería y el valor `false`.
- `void ordenar_AF(tListaCorreos &correos)`: Dada una lista de correos la devuelve ordenada por asunto y fecha. Como es una clave de ordenación doble, habrá que redefinir el operador de comparación en el módulo que corresponda.

## Módulo ListaRegistros

Declara dos tipos de datos `tRegistro` y `tListaRegistros` para gestionar los registros y las listas de registros. El tipo `tRegistro` debe tener los siguientes campos:

- Un identificador **único** (string). Para que el identificador sea único, se compone de la concatenación del emisor y la fecha (p.e. *pepe@fdimail.com\_143456443*). Este identificador coincide con el identificador del `tCorreo`.
- Un booleano que indica si el correo ha sido leído o no (por el propietario del registro).

El tipo `tRegistro` es tan sencillo que no tiene operaciones asociadas.



La lista `tListaRegistros` será una lista de tamaño variable de registros. Es **IMPORTANTE** que tengas en cuenta que esta lista no tiene ningún orden especial, sino que se encuentra **ordenada según “el orden de llegada” de los registros**.

Define al menos los siguientes subprogramas:

- `void inicializar(tListaRegCorreo &listaReg)`: Inicializa la lista.
- `void cargar(tListaRegCorreo &listaReg, ifstream& archivo)`: Dado un flujo de archivo de entrada (ya abierto), lee los datos que corresponden a una lista de registros y la devuelve.
- `void guardar(const tListaRegCorreo &listaReg, ofstream& archivo)`: Dado un flujo de archivo de salida (ya abierto), guarda los datos de la lista de registro.
- `bool insertar(tListaRegCorreo &listaReg, tRegCorreo registro)`: Dado un registro lo inserta al final de la lista. Si la lista está llena devuelve `false`, en otro caso devuelve `true`. Este subprograma se ejecutará cuando un usuario envíe un correo ya que se insertará el registro correspondiente en la lista de registros que representa su bandeja de salida, y también en las listas de registros que representan las bandejas de entrada de cada uno de los destinatarios del correo.
- `bool borrar(tListaRegCorreo &listaReg, string id)`: Dado un identificador de correo, busca el registro correspondiente y si lo encuentra lo elimina de la lista (¡sin dejar huecos!). Si no lo encuentra, devuelve `false`, en otro caso devuelve `true`. Este subprograma representa la acción de un usuario del borrado de un determinado correo de una de sus bandejas. **OJO**: esta operación sólo supone que el registro es borrado de la lista de registros, pero el correo seguirá existiendo en la lista de correos.
- `bool correoLeido(tListaRegCorreo &listaReg, string id)`: Dado un identificador de correo, busca el correspondiente registro y pone el indicador de leído a `true`. La operación devuelve un booleano indicando si se encontró o no el registro.
- `int buscar(const tListaRegCorreo &listaReg, string id)`: Dado un identificador de correo y la lista, devuelve, si dicho identificador existe en la lista, su posición, y si no existe devuelve `-1`.

## Módulo Usuario

Declara un tipo de datos `tUsuario` para guardar la información de un correo:

- El nombre o identificador del usuario (`string`). Debe ser único en el gestor, es decir, en la lista de usuarios del gestor.
- La contraseña del usuario (`string`).
- La lista de registros de mensajes recibidos que será de tipo `tListaRegistros` y que nos permite implementar la bandeja de entrada del usuario.
- La lista de registros de mensajes enviados que será de tipo `tListaRegistros` y que nos permite implementar la bandeja de salida del usuario.

Implementa, al menos, los siguientes subprogramas:

- `bool cargar(tUsuario& usuario, ifstream& archivo)`: Dado un flujo de archivo (ya abierto), se carga un usuario de fichero.
- `void guardar(const tUsuario& usuario, ofstream& archivo)`: Dado un flujo de archivo (ya abierto), se guarda un usuario en fichero.
- `void inicializar(tUsuario& usuario, string id, string contrasenia)`: Recibe un identificador de usuario y una contraseña y los asigna al usuario.
- `bool validarContrasenia(const tUsuario &usuario, string contrasenia)`: Recibe una contraseña y un usuario y devuelve si la contraseña es correcta o no.

## Módulo ListaUsuarios

Declara un tipo de datos `tListaUsuarios` para gestionar la lista de usuarios. Es **IMPORTANTE** que tengas en cuenta que esta lista se encuentra **ordenada por el identificador del usuario**. Debes implementarla como una lista de tamaño variable.

Los subprogramas básicos que debe tener la lista son los siguientes:

- `void inicializar(tListaUsuarios &usuarios)`: Inicializa la lista.
- `bool cargar(tListaUsuarios& usuarios, string dominio)`: Implementa la carga de la lista de usuarios desde el fichero de usuarios de nombre `<NombreDominio>_usuarios.txt`.
- `void guardar(const tListaUsuarios& usuarios, string dominio)`: Implementa el guardado de la lista de usuarios en el fichero de usuarios de nombre `<NombreDominio>_usuarios.txt`.
- `bool anadir(tListaUsuarios& usuarios, const tUsuario& usuario)`: Añade un usuario en la posición de la lista que le corresponde, si hay sitio para ello. Además devuelve un booleano indicando si la operación tuvo éxito o no.

- `bool buscarUsuario(const tListaUsuarios& usuarios, string id, int& posicion)`: Dado un identificador de usuario y la lista, devuelve, si dicho identificador existe en la lista, su posición y el valor `true`, y si no existe en la lista, la posición que le correspondería y el valor `false`.

## Módulo Gestor

Declara un tipo de datos `tGestor` para guardar la información de un correo:

- El dominio del gestor de correo (`string`). Se trata de un identificador como por ejemplo “`fdimail.com`”. Nos indica la lista de correos y de usuarios que debemos cargar.
- La lista de correos del sistema que será de tipo `tListaCorreos` y que guarda la única copia de los correos que existirá en la aplicación.
- La lista de usuarios del sistema que será de tipo `tListaUsuarios`.
- Índice del usuario activo en el sistema (de tipo `int`). Nos indica la posición en la lista de usuarios que ocupa el usuario que se encuentra registrado en el sistema. Si no hay ningún usuario registrado (o el usuario ha cerrado sesión) valdrá `-1`.

El módulo gestor debe implementar, al menos, los siguientes subprogramas:

- `bool arrancar(tGestor& gestor, string dominio)`: Inicializa el gestor e intenta arrancarlo cargando la información del dominio que se le pasa como parámetro. Para ello inicializará y cargará la lista de usuarios y de correos de dicho dominio. Si tiene éxito en todas las operaciones devuelve `true` y si alguna falla devuelve `false`.
- `void apagar(const tGestor &gestor)`: Esta operación apaga el gestor y guarda para ello las listas de usuarios y de correos del dominio que se encontrase activo en ese momento.
- `bool crearCuenta(tGestor &gestor)`: Lee los datos de usuario necesarios para crear una cuenta (id y contraseña) y si el id de usuario no existe y hay espacio en la lista de usuarios, crea la cuenta del usuario y registra al nuevo usuario como usuario activo en el gestor. Devuelve un booleano indicando si la operación tuvo éxito.
- `bool iniciarSesion(tGestor &gestor)`: Lee los datos de usuario necesarios para validar la cuenta (id y contraseña) y si el usuario existe y la contraseña coincide, registra al usuario como usuario activo. Devuelve un booleano indicando si la operación tuvo éxito.

- `void leerCorreo(tGestor& gestor, tListaRegCorreo& listaReg):` Solicita el correo que el usuario quiere leer (será el número con el que el correo es mostrado por pantalla en la bandeja correspondiente), valida que existe y si es así, lo marca como correo leído. A continuación, busca el correo en la lista de correos y si lo encuentra lo muestra en la pantalla de lectura del correo (Fig. 3).
- `void enviarCorreo(tGestor& gestor, const tCorreo &correo):` Este subprograma implementa el envío del correo en nuestra aplicación. Para ello el correo recibido como parámetro es insertado en la lista de correo. Si ha habido éxito, entonces se inserta el registro correspondiente en la lista de registros enviados del emisor y se intenta insertar igualmente un registro de dicho correo en la lista de registros recibidos del destinatario del correo. Si el destinatario no existe o si su bandeja de entrada está llena, entonces se mostrará un mensaje de error.
- `void borrarCorreo(tGestor& gestor, tListaRegCorreo& listaReg):` Este subprograma implementa el borrado del correo de una bandeja de un determinado usuario (**OJO**: el correo no se borra de la lista de correos). Para ello, solicita el correo que el usuario quiere borrar (será el número con el que el correo es mostrado por pantalla en la bandeja correspondiente), valida que existe y si es así, procede a borrarlo.
- `void lecturaRapida(tGestor& gestor, tListaRegCorreo& listaReg):` Este subprograma implementa la lectura rápida de correos sin leer. El resultado es que muestra en una pantalla todos los correos sin leer ordenados por asunto (ignorando todos los "Re: ") y por fecha. Al finalizar su ejecución los correos sin leer quedarán marcados como leídos.

## Módulo Principal

Habrás además un fichero .cpp adicional que contendrá el *main* de la aplicación y que incluirá todos los módulos que necesite para su correcto funcionamiento.

### 3. Detalles de implementación

A la hora de completar la práctica, deberás seguir las siguientes indicaciones.

#### 3.1. Tipos de datos especiales

##### **stringstream**

En C++ es posible operar con un string como lo hacemos con los flujos de E/S. En esta práctica puede resultarnos muy útil generar strings como si estuviéramos mandando información a un flujo de salida, ya sea la salida estándar `cout` o un flujo de fichero `ofstream`. En lugar de dar una explicación usaremos un ejemplo:

```
#include <sstream> // Es necesario incluir la biblioteca sstream
...
string resultado;
string nombre= "Pepe";
stringstream flujo; // Flujo
flujo << "Hola " << nombre << endl; // Pasamos datos al flujo
resultado=flujo.str(); // Generamos un string
// Al final resultado contendrá la cadena "Hola Pepe\n"
```

##### **Fecha**

Las fechas se representarán en el formato UNIX, es decir, como un entero con el número de segundos transcurridos desde el 1 de enero de 1970. Debes por tanto declarar el tipo `tFecha` mediante:

```
typedef time_t tFecha;
```

En la función de creación de un correo nuevo se obtendrá la fecha actual del sistema:

```
correo.fecha = time(0); → será necesario incluir la librería ctime
```

La lectura/escritura de la fecha de/en fichero se realiza usando el operador habitual de extracción/inserción de los enteros.

Deberás definir además la siguiente función para mostrar la fecha en formato Año/Mes/Día, Hora/Mins/Segs:

```
string mostrarFecha(tFecha fecha){
    stringstream resultado;
    tm ltm;
    localtime_s(&ltm, &fecha);
    resultado << 1900 + ltm.tm_year << "/" << 1 + ltm.tm_mon << "/" << ltm.tm_mday;
    resultado<< " (" << ltm.tm_hour << ":" << ltm.tm_min << ":" << ltm.tm_sec << ")";
    return resultado.str();
}
```

Para mostrar sólo el día de la fecha en formato Año/Mes/Día (sin hora):

```
string mostrarSoloDia(tFecha fecha){
    stringstream resultado;
    tm ltm;
    localtime_s(&ltm, &fecha);
    resultado << 1900 + ltm.tm_year << "/" << 1 + ltm.tm_mon << "/" << ltm.tm_mday;
    return resultado.str();
}
```

### 3.2. Ficheros de datos

La Figura 7 muestra el contenido de un fichero de ejemplo de nombre genérico <NombreDominio>\_usuarios.txt que contiene la lista de usuarios del sistema ordenada por orden alfabético del identificador de usuario y que acaba con el centinela XXX.



Figura 7. Ejemplo de fichero con una lista de usuarios

Cada usuario tiene sus campos: nombre, contraseña, bandeja de entrada y bandeja de salida. Ambas bandejas son del tipo `tListaRegistros`. Cada lista de registros se compondrá en primer lugar del valor del contador y a continuación de tantos datos de tipo `tRegistro` como indique el contador (si el contador vale 0, no habrá ningún registro). El `tRegistro` se compone de su identificador de correo y del booleano que indica si el mensaje está leído o no ambos en la misma línea.

Por otro lado, existe un fichero <NombreDominio>\_listaCorreo.txt que guarda todos los correos gestionados por el sistema. En la Figura 8 se muestra un ejemplo de dicho fichero. Como se puede ver se compone de correos ordenados por el identificador de correo y acaba con el centinela XXX.

```

correo {
  arturo@fdimail.com 1426614381
  1426614381
  arturo@fdimail.com
  ginebra@fdimail.com
  Mensaje de prueba
  Hola,
  Parece que ya tenemos correo en Camelot!
  Arturo
  X
}

correo {
  arturo@fdimail.com_1426614458
  1426614458
  arturo@fdimail.com
  lancelot@fdimail.com
  Comprar mesa de reuniones
  Hola,
  he pensado que podíamos comprar una mesa de forma redonda
  para las reuniones. ¿Te parece buena idea?
  Arturo
  X
}

correo {
  lancelot@fdimail.com_1426613678
  1426613678
  lancelot@fdimail.com
  arturo@fdimail.com
  Cazamos mañana?
  Hola!

  ¿Salimos a cazar mañana?
  Lancelot
  X
  XXX
}

```

Figura 8. Ejemplo de fichero con la lista de correos

Cada correo se compone de varias líneas: identificador, fecha, emisor, receptor, asunto y cuerpo del mensaje. El cuerpo de un mensaje está compuesto de varias líneas. Esto hará que necesites crear unos procedimientos para leer el cuerpo de teclado y para cargarlo.

Para indicar el fin de un correo usaremos un centinela X. Esto nos permite saber también cuándo acaba el cuerpo del mensaje.

### 3.3. Cargar y guardar

Para realizar la carga y el guardado de los ficheros es **MUY IMPORTANTE** que tengas en cuenta lo siguiente: la carga y guardado en ficheros NO se realiza en un único módulo. En su lugar, cada módulo se encarga de cargar/guardar los datos que le corresponden.

Para que esto sea posible se tienen que dar dos condiciones:

- La apertura y cierre de un fichero se realiza en un único sitio: el subprograma que inicia la operación de cargar/guardar.

- El subprograma coordina la operación invocando al resto de subprogramas de cargar/guardar los cuales reciben el fichero (ifstream / ofstream) abierto **como parámetro por referencia** para cargar/guardar lo que corresponda.

Vamos a poner un ejemplo con el cargar la lista de usuarios desde fichero. La operación de cargar la lista de usuarios se inicia con la invocación al subprograma cargar del módulo de la lista de usuarios:

```
bool cargar(tListaUsuarios& usuarios, string dominio)
```

Este subprograma intentará abrir el fichero <dominio>\_usuarios.txt y si lo consigue invocará para cada usuario al subprograma que gestiona la carga de un usuario que estará en el módulo usuario:

```
bool cargar(tUsuario& usuario, ifstream& archivo)
```

Como podemos ver, a este subprograma habrá que pasarle como parámetro de E/S el flujo de entrada que hemos obtenido al abrir el fichero. Este subprograma leerá los campos básicos del usuario en orden (identificador y contraseña), sin embargo, cuando llegue el turno de leer la bandeja de entrada y la bandeja de salida (que son de tipo tListaRegistros) tendrá que realizar dos invocaciones consecutivas al subprograma cargar del módulo de la lista de registros:

```
void cargar(tListaRegCorreo &listaReg, ifstream& archivo)
```

De nuevo a este subprograma habrá que pasarle como parámetro de E/S el flujo de entrada que recibió el cargar del módulo usuario.

### 3.4. Indicaciones importantes

- Las listas de tamaño variable deberán implementarse con la estructura de array + contador vista en el Tema 6.
- Siempre que se pueda, las operaciones de búsqueda serán binarias.
- Todos los módulos se implementarán con archivos .h y .cpp. En el archivo .h se incluirán los tipos de datos del módulo y los prototipos de las funciones públicas con comentarios describiendo qué hace cada función, qué datos recibe y qué datos devuelve.
- Los módulos ListaUsuarios, ListaCorreos y ListaRegistros únicamente realizan operaciones con listas, y nunca deben escribir nada en pantalla. Pueden devolver valores al programa que los llame para que éste escriba en pantalla lo que sea necesario.



## 4. Partes opcionales

### 4.1. Posibilidad de múltiples destinatarios

Se puede hacer más realista la aplicación permitiendo múltiples destinatarios para un correo. Para ello se puede definir una lista de tamaño variable de `string` para los destinatarios de un correo.

### 4.2. Eliminación de correos en el gestor

Tal y como se borran los correos, puede ocurrir que un correo del gestor haya sido borrado tanto por su emisor como por su destinatario, y sin embargo seguirá en la lista de correos del gestor sin estar ya “apuntado” por ningún registro de ningún usuario. Como esto puede provocar problemas de espacio a largo plazo, se propone realizar una eliminación de correos no apuntados por ningún usuario.

Para ello, cuando un usuario borra un correo de su bandeja, si ese usuario era el último usuario que tenía en su bandeja el correo, se procederá a eliminar el correo de la lista de correo (además de eliminar el registro de dicho correo de la bandeja del usuario).

## 5. Entrega de la práctica

La práctica se entregará a través del Campus Virtual. Se habilitará una nueva tarea **Entrega de la Práctica 4** que permitirá subir un único archivo. Hay que subir un archivo comprimido `Practica4GrupoXX.zip` que contenga los archivos `.h` y `.cpp` del proyecto.

Fin del plazo de entrega: **17 de mayo a las 23:55.**