UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
Año 2017 – 2do cuatrimestre

ALGORITMOS Y PROGRAMACIÓN I (95.11)
CÁTEDRA: KHUM

TRABAJO PRÁCTICO N.º 3
TEMA: Final
FECHA: 02/12/2017

INTEGRANTES:
Cabrera Mauricio - # 101334
<maurihuergo@hotmail.com>
Floutard, Aurelien - # 101999
<aure.flout@hotmail.fr>

Trabajo Práctico Final

Algoritmos y Programación I (95.11/75.02)

13 de noviembre de 2017

1. Objetivo

El objetivo del presente trabajo es diseño y desarrollo de una aplicación en modo consola, escrita en lenguaje ANSI-C89, que permita cargar una "red social" *ad-hoc* y realizar algunas operaciones con la misma.

2. Alcance

Mediante el presente TP se busca que el/la estudiante adquiera y aplique, conocimientos sobre los siguientes temas, además de los vistos anteriormente:

- Directivas al preprocesador C
- Programas en modo consola
- Tipos enumerativos
- Funciones
- Salida de datos con formato
- Modularización
- Arreglos/Vectores
- Memoria dinámica
- Arg. en línea de comandos
- Estructuras
- Archivos
- Punteros a función
- Modularización
- Tipos de Dato Abstracto, particularmente contenedores

Fecha límite de entrega: 4 de diciembre de 2017

3. Introducción

3.1. Red Social

En nuestra aplicación, la red social está formada por *usuarios*. Cada usuario tiene un *identificador* dentro de nuestra "base de datos"[1], un *usuario*, un *nombre*, un registro con sus *amigos* y un registro con *mensajes*.

En particular, un usuario será representado en la aplicación utilizando la siguiente estructura de datos:

```
1 struct usuario {
2     int id;
3     t_cadena nombre;
4     t_cadena usuario;
5     vector_s amigos;
6     lista_s  mensajes;
7 };
```

donde `t_cadena` es un alias del tipo `char *`, y `vector_s` y `lista_s` son contenedores: tipos de datos abstractos para almacenar datos, ambos polimórficos. Particularmente, el vector `amigos` contiene los identificadores de los usuarios (campo id), y la lista `mensajes` almacena otro tipo de dato: un *mensaje*. La figura 1 muestra una representación gráfica de un usuario cargado.

Los mensajes son estructuras que contienen un identificador, una fecha (*stamp*), un mensaje (que puede tener un largo máximo fijo, por ejemplo, 140 carácteres), y el *id* de quien lo publica.

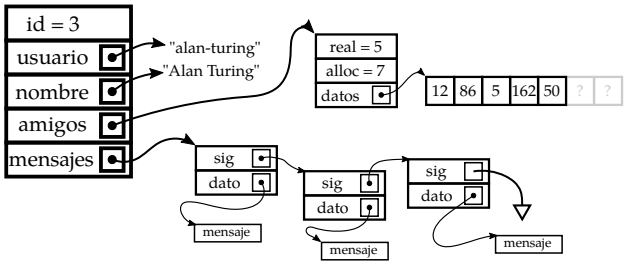


Figura 1: Diagrama de la estructura de datos correspondiente a un usuario

3.2. Archivo de configuración

El o los archivos donde se almacenan los usuarios de la red social se pueden pensar como un archivo de configuración INI. El mismo contiene secciones que comienzan con una línea que contienen una palabra entre corchetes, por ejemplo: `[mariano]`. Cada sección se corresponde con los datos de un usuario, en donde el nombre de la sección es el nombre de usuario correspondiente a la sección. Luego se encuentran los datos del usuario. La lista de identificadores válidos para cada dato del usuario es: `id`, `nombre`, `amigos`, y `mensaje`. Cada línea que contenga datos está conformada por un identificador de la lista anterior, el símbolo `=`, seguido de los datos, como se muestra a continuación:

[1] en la realidad se utilizan bases de datos para almacenar la información, pero no será el caso en este trabajo

```
identificador = datos
```

3.2.1. Identificador de datos: usuario

- Se identifica con el título de la sección.
- Debe ser una cadena de carácteres.
- Se aceptan todos los carácteres imprimibles.
- Los usuarios NO pueden estar repetidos.

3.2.2. Identificador de datos: id

- Debe ser un número positivo (> 0).
- Sólo puede haber un campo *id* por usuario.

3.2.3. Identificador de datos: nombre

- Se acepta cualquier secuencia de carácteres.
- Sólo puede haber un campo *nombre* por usuario.

3.2.4. Identificador de datos: amigos

- Contiene números positivos separados por comas.
- Cada número representa el *id* de un usuario.
- Lo usuarios pueden NO existir en la red.

3.2.5. Identificador de datos: mensaje

- Es el único campo que puede aparecer repetido.
- Es una línea con campos separados por comas (ver TP2).
    - El primer campo es el *id* del mensaje.
    - El segundo dato es el *timestamp* del mensaje, en formato conforme al estándar ISO 8601: `AAAA-MM-DD`.
    - El tercer campo es el *id* de un usuario, que puede NO existir en la red.
    - El cuarto campo es el mensaje en sí, formado por una secuencia de carácteres.

Ejemplo de archivo válido

```
[mariano]
id = 23
nombre = Mariano Moreno
amigos = 2,15,101,36
mensaje = 1965,1809-04-15,23,Quiero más una libertad peligrosa que una se
mensaje = 2069,1811-09-15,2,Hacía falta tanta agua para apagar tanto fueg

[cornelio]
id = 2
nombre = Cornelio Saavedra
amigos = 263,15,23
mensaje = 2069,1811-09-15,2,Hacía falta tanta agua para apagar tanto fueg
```

Figura 2: Ejemplo de archivo con los datos de la red social

La lectura de archivos de un proceso que consume demasiados recursos, por lo que sólo se puede recorrer el archivo una vez, de principio a fin.

4. Desarrollo

La aplicación a desarrollar debe procesar el archivo de configuración recibido y crear en memoria una lista de usuarios, que conforma la red social. Para ello, es necesario contar con tipos de datos contenedores, a saber: lista y vector. Los mismos deben ser desarrollados por el grupo de trabajo, probados y validados. Se deben agregar, además, estructuras para el manejo de usuarios y mensajes.

Con todas las estructuras de datos probadas y validadas, se debe crear una función que cargue de un archivo de configuración, todos los usuarios contenidos en el mismo. Se recomienda contemplar el caso del uso de una cantidad indeterminada de archivos de configuración. Una vez terminado el módulo de procesamiento del archivo, se puede realizar una prueba con el archivo de la figura 2. En caso de ejecución correcto, se debe crear una estructura similar a la de la figura 3.

Al disponer de los datos en memoria, se puede continuar con el desarrollo de la aplicación.

Si bien hay una gran cantidad de operaciones que se pueden realizar, en este caso sólo se implementará la eliminación de usuarios.

4.1. Manejo de la red social

La aplicación desarrollada debe ser invocable por línea de comandos de acuerdo con el siguiente ejemplo

```
./red_social --eliminar <filtro> -o <fmt> [archivo1 [archivo2 ...]]
./red_social  -e        <filtro> -o <fmt> [archivo1 [archivo2 ...]]
```

donde:

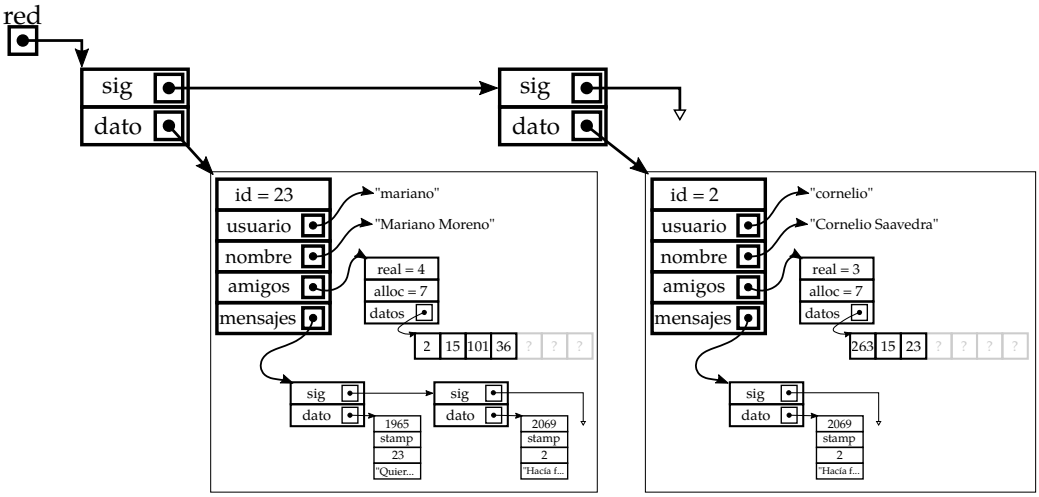


Figura 3: Representación gráfica del archivo de la figura 2

eliminar indica qué se desea eliminar. En este caso, sólo se eliminarán usuarios. Los usuarios podrán ser eliminados según el identificador o el nombre de usuario. Si se desea eliminar por identificador, el filtro será `i:numero` donde número es el identificador. Si se desea eliminar por nombre de usuario, el filtro debe ser `u:usuario`.

A modo de ejemplo, si en la figura 3 se quisiera eliminar de la red al usuario "Mariano Moreno", el programa podría ser invocado como:

```
./red_social --eliminar i:23       ...
./red_social  -e        u:mariano  ...
```

Esta opción también admite ser invocada con una opción corta, que es `-e` en lugar de `--eliminar`.

output indica cómo debe ser la salida y puede tener 2 opciones: *single* y *multi*. Si la opción es *single* entonces se debe imprimir el resultado por pantalla, siguiendo el formato del archivo de configuración. Si la opción es *multi* entonces se debe imprimir cada usuario en un archivo diferente, cuyos nombres pueden ser: `"%i-%s"`, donde `%i` es el id del usuario y `%s` es el usuario.

Esta opción también admite ser invocada con una opción corta, que es `-o` en lugar de `--output`.

5. Testing

La aplicación y sus modulos deben ser testeados. Es necesario escribir aplicaciones rudimentarias para hacer pruebas a los módulos. Por ejemplo, el siguiente código –con hardcodes– valida la creación de un vector, la inserción de datos, la toma de datos, y la destrucción del vector.

```
1 int main(void)
```

```
2  {
3      vector_s * v;
4      int datos[] = {2, 15, 101, 36};
5      int i, largo;
6
7      if((v = VECTOR_crear(2)) == NULL)
8          return EXIT_FAILURE;
9
10     for(i = 0; i < sizeof(datos) / sizeof(datos[0]); i++)
11     {
12         if(VECTOR_insertar(v, &datos[i]) != true)
13         {
14             fprintf(stderr, "%s\n", "ERROR: fallo la
                   insercion");
15             VECTOR_destruir(&v, NULL);
16             return EXIT_FAILURE:
17         }
18     }
19
20     for(i = 0, largo = VECTOR_largo(v); i < largo; i++)
21         printf("v[%i] = %i\n", i, *(int *)VECTOR_get(v, i)
               );
22
23     VECTOR_destruir(&v, NULL);
24
25     return EXIT_SUCCESS;
26 }
```

5.1. Fugas de memoria

Las fugas de memoria de la aplicación pueden ser advertidas utilizando la aplicación *valgrind*. Para que la misma indique dónde se producen fugas de memoria, es necesario compilar nuestra aplicación en modo debug.

Una vez compilada la aplicación, valgrind se ejecuta de la siguiente manera:

```
valgrind ./test_vector_2
valgrind ./test_vector_3 argumento1 argumento2 ...
```

Al ejecutar correctamente el comando Y si el programa no tiene fallas, se ve una leyenda similar a la siguiente:

```
==23390== HEAP SUMMARY:
==23390==     in use at exit: 0 bytes in 0 blocks
==23390==   total heap usage: 89,408 allocs, 89,408 frees, 5,057,256
    bytes allocated
==23390==
==23390== All heap blocks were freed -- no leaks are possible
==23390==
==23390== For counts of detected and suppressed errors, rerun with:
   -v
==23390== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
   from 0)
```

Ejemplo 1: Resultado de ejecución de valgrind sin fugas de memoria

Si en cambio el programa tiene fugas de memoria, el mensaje es:

```
==25854== HEAP SUMMARY:
==25854==     in use at exit: 456,722 bytes in 3,932 blocks
==25854==   total heap usage: 89,408 allocs, 85,476 frees, 5,057,256
    bytes allocated
==25854==
==25854== LEAK SUMMARY:
==25854==    definitely lost: 456,722 bytes in 3,932 blocks
==25854==    indirectly lost: 0 bytes in 0 blocks
==25854==      possibly lost: 0 bytes in 0 blocks
==25854==    still reachable: 0 bytes in 0 blocks
==25854==         suppressed: 0 bytes in 0 blocks
==25854== Rerun with --leak-check=full to see details of leaked
    memory
==25854==
==25854== For counts of detected and suppressed errors, rerun with:
    -v
==25854== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
    from 0)
```

Ejemplo 2: Resultado de ejecución de valgrind con fugas de memoria

6. Restricciones

La realización de los programas pedidos está sujeta a las siguientes restricciones:

- Debe realizarse en grupos de **3 (tres)** integrantes.
- No está permitida la utilización de `scanf()`, `gets()`[2], `fflush(stdin)`[3],la biblioteca `conio.h`[4] (`#include <conio.h>`), etc.
- Debe recurrirse a la utilización de funciones mediante una adecuada parametrización.
- Deben utilizarse punteros a función en la eliminación del usuario, por lo menos.
- Deben utilizarse TDAs para el vector y las listas, por lo menos.
- No está permitido en absoluto tener hard-codings:

```
1 ...
2 char c;
3 ...
4 if (c == 'R')                    /* ¡¡hard-coded!!*/
5     printf("%s","xxxxxxxx");     /* ¡¡hard-coded!!*/
6 else if(c == 'A')                /* ¡¡hard-coded!!*/
7     printf("%s","yyyyyyyy");     /* ¡¡hard-coded!!*/
8 ...
```

[2]obsoleta en C99 [3], eliminada en C11 [4] por fallas de seguridad en su uso.
[3]comportamiento indefinido para flujos de entrada ([3],[4]). Definida en estándar POSIX.
[4]biblioteca no estándar, con diferentes implementaciones y licencias, y no siempre disponible.

sino que debe recurrirse al uso de `ETIQUETAS`, `CONSTANTES SIMBÓLICAS`, `MACROS`, etc.

Los ejemplos no son exhaustivos, sino que existen otros hard-codings y tampoco son aceptados.

- Hay ciertas cuestiones que no han sido especificadas intencionalmente en este Requerimiento, para darle al/la desarrollador/a la libertad de elegir implementaciones que, según su criterio, resulten más convenientes en determinadas situaciones. Por lo tanto, se debe explicitar cada una de las decisiones adoptadas, y el o los fundamentos considerados para las mismas.

7. Entrega

La fecha límite de entrega del trabajo práctico es el 4/12.

No se requiere entrega en papel, pero se acepta. Deberá realizarse una entrega digital, a través del campus de la materia, de un único archivo cuyo nombre debe seguir el siguiente formato:

```
YYYYMMDD_apellido1-apellido2-apellido3-N.tar.gz
```

donde `YYYY` es el año (2017), `MM` el mes y `DD` el día en que uno de los integrantes sube el archivo, `apellido-1a3` son los apellidos de los integrantes ordenados alfabéticamente, `N` indica el número de vez que se envía el trabajo (`1`, `2`, etc.), y `.tar.gz` es la extensión, que no necesariamente es `.tar.gz`.

El archivo comprimido debe contener los siguientes elementos:

- La correspondiente documentación de desarrollo del TP (en formato pdf), siguiendo la numeración siguiente, incluyendo:

  1. Carátula del TP. Incluir una dirección de correo electrónico.
  2. Enunciado del TP.
  3. Estructura funcional de los programas desarrollados.
  4. Explicación de cada una de las alternativas consideradas y las estrategias adoptadas.
  5. Resultados de la ejecución (corridas) de los programas, captura de las pantallas, bajo condiciones normales e inesperadas de entrada.
  6. **Archivos de prueba utilizados.**
  7. Reseña sobre los problemas encontrados en el desarrollo de los programas y las soluciones implementadas para subsanarlos.
  8. Bibliografía (ver aparte).
  9. Indicaciones sobre la compilación de lo entregado para generar la aplicación.

     NOTA: Si la compilación del código fuente presenta mensajes de aviso (warning), notas o errores, los mismos deben ser comentados en un apartado del informe.

NOTA: El Informe deberá ser redactado en *correcto* idioma castellano.

- Códigos fuentes en formato de texto plano (.c y .h), *debidamente documentados*.

NOTA: Todos los integrantes del grupo deben subir el *mismo* archivo.
NOTA: Se debe generar y subir un único archivo (comprimido) con todos los elementos de la entrega digital. **NO usar RAR.** La compresión RAR no es un formato libre, en tanto sí se puede utilizar *ZIP*, *GUNZIP*, u otros (soportados, por ejemplo, por la aplicación de archivo *TAR*).

Si no se presenta cada uno de estos ítems, será rechazado el TP.

8. Bibliografía

Debe incluirse la referencia a toda bibliografía consultada para la realización del presente TP: libros, artículos, URLs, etc., citando:

- Denominación completa del material (Título, Autores, Edición, Volumen, etc.).
- Código ISBN del libro (opcional: código interbibliotecario).
- URL del sitio consultado. No poner Wikipedia.org o stackexchange.com, sino que debe incluírse un enlace al artículo, hilo, etc. consultado.

Utilizando LaTeX, la inclusión de citas/referencias es trivial. Los editores de texto gráficos de las suites de ofimática, como LibreOffice Write o MS Word, admiten plugins que facilitan la inclusión.

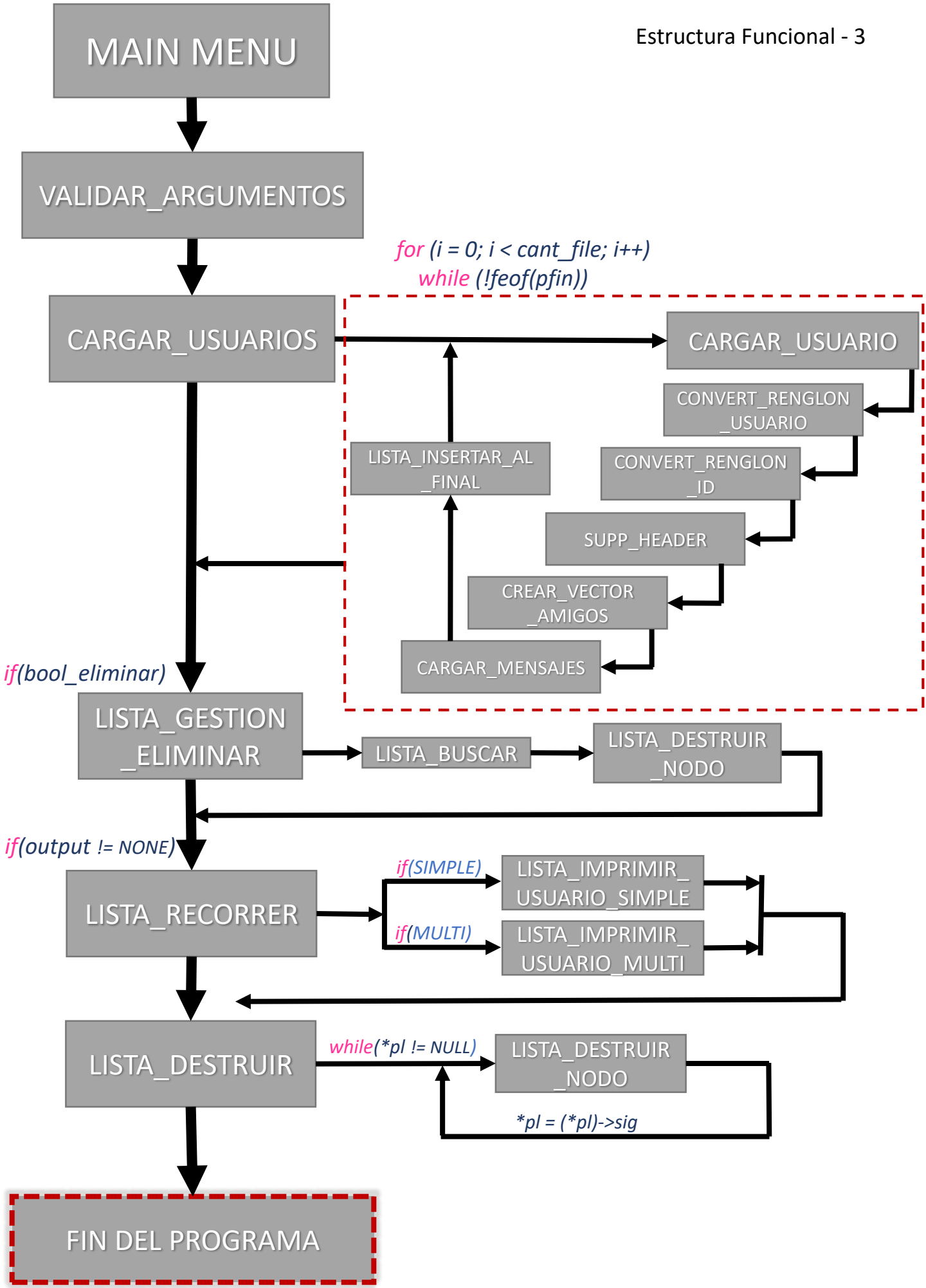
Ejemplo de referencias

[1] B.W. Kernighan y D.M. Ritchie. *The C Programming Language*. 2.ª ed. Prentice-Hall software series. Prentice Hall, 1988. ISBN: 9780131103627.

[2] P. Deitel y H. Deitel. *C How to Program*. 7.ª ed. Pearson Education, 2012. ISBN: 9780133061567.

[3] ISO/IEC. *Programming Languages – C*. ISO/IEC 9899:1999(E). ANSI, dic. de 1999, págs. 270-271.

[4] ISO/IEC. *Programming Languages – C*. INCITS/ISO/IEC 9899:2011. INCITS/ISO/IEC, 2012, pág. 305.


MAIN MENU
VALIDAR_ARGUMENTOS
for (i = 0; i < cant_file; i++)
while (!feof(pfin))
CARGAR_USUARIOS
CARGAR_USUARIO
CONVERT_RENGLON _USUARIO
CONVERT_RENGLON _ID
SUPP_HEADER
CREAR_VECTOR _AMIGOS
CARGAR_MENSAJES
LISTA_INSERTAR_AL _FINAL
if(bool_eliminar)
LISTA_GESTION _ELIMINAR
LISTA_BUSCAR
LISTA_DESTRUIR _NODO
if(output != NONE)
LISTA_RECORRER
if(SIMPLE)
if(MULTI)
LISTA_IMPRIMIR_ USUARIO_SIMPLE
LISTA_IMPRIMIR_ USUARIO_MULTI
LISTA_DESTRUIR
while(*pl != NULL)
LISTA_DESTRUIR _NODO
*pl = (*pl)->sig
FIN DEL PROGRAMA

Alternativas Consideradas y Estrategias Adoptadas

Al iniciar el desarrollo del programa tuvimos en cuenta la formación de las estructuras para desarrollar todo el TP. Luego vimos las funciones que dio el profesor en la practica y la forma ordena de generalizar las funciones como listas y nodos para luego usarse a conveniencia así que adaptamos las funciones que hicimos en al principio para que funciones de esta temática.

También otro cambio que hicimos en el sistema fue el uso mucho mas generalizado con las TDA, pero hubo muchos inconvenientes con su uso más que había fugas de memoria que no sabíamos de dónde venían así que limitamos mas el uso de estas.

Uno de los cambios también fue el permitir que la función ande con al menos un argumento. Al principio consideramos que si o si para que funcione el programa necesitaba de 5 argumentos, los cuales eran, el flag de eliminar y output junto a sus formatos y al menos un archivo donde tomar los datos de los usuarios, pero luego nos dimos cuenta de que podía ser posible que el usuario podía elegir en que se elimine un archivo de la red pero que no lo muestre o simplemente mostrar la red sin ningún cambio en particular, y luego llegamos a la conclusión de que hasta que el usuario podía elegir simplemente poner un archivo y ejecutar el programa aunque no muestre absolutamente nada.

El las funciones en general de todo el programa están hechos para que nos entreguen un valor de estado, de esta forma corroboramos si la función se ejecutó con éxito o si hubo un problema inesperado en medio del proceso, por esta razón la gran mayoría de las modificaciones de las variables se hacen por referencia.

También la separación de las funciones en dos archivos distintos, uno ellos es LISTAS.c el cual tiene todas las funciones relacionadas con el manejo de listas, donde se busque, se crean y se destruyen los nodos y las listas pertinentes al programa. Y el otro archivo es funciones.c, el cual tiene el manejo de los argumentos de y los archivos, las funciones relacionadas con este transforman las cadenas de caracteres que hay en el o los archivo/s usado/s en la ejecución del programa en el tipo de dato necesario para el funcionamiento del programa.

Descripción de las funciones usada para el programa

Funciones.c

Cargar_usuarios

Como dice se nombre carga todos los usuarios en del archivo usando la función cargar_usuario hasta que se llega al final del archivo.

Cargar_usuario

Carga un solo nodo a la lista usuario, esta función es la que se encarga de todo el proceso de extracción de información y administración, esta se apoya en varias subfunciones que se describen a continuación.

Conver_renglon_usuario

Función auxiliar usada para extraer de la cadena de caracteres pasada el nombre de usuario.

Convert_renglón_id

Convierte la cadena de caracteres pasada en a simplemente el ID del usuario.

Cinvert_vector_amigos

Se ingresa una cadena de caracteres en el termina l y devuelven los datos de la cantidad de amigos.

Numero_amigos

Cuenta la cantidad de amigos que tenemos.

Supp_header

Ayuda a reconocer el signo igual del texto para después guardar el dato que sigue después del signo en una cadena de caracteres.

Cargar_mensajes

Introduce en una lista los mensajes que tiene el usuario.

Cargar_mensaje

Esta es una subfunción de la función anterior que se encarga cada nodo de mensaje.

Leer_mensaje

Extrae cada parte del mensaje y lo guarda en una cadena de caracteres.

Validar_argumentos

Verifica que los argumentos cumplan con lo necesario para poder correr el programa, una vez verificados los procesa para que el programa funciones según lo indicado en los flags y abriendo la cantidad necesaria de archivos.

Imprimir_estado

Imprime el estado de la situación, este solo aparece cuando hay un error o al finalizar el programa si todo se completó exitosamente.

LISTA.c

LISTA_vacia

Chequea si la lista esta vacía o no, devolvió un 0 si no lo esta o un 1 si está vacía.

LISTA_crear_nodo

Crea un nodo para ocupar un usuario o un mensaje.

LISTA_destruir_nodo

Libera el espacio de memoria pedido para el nodo y luego hace que apunte a NULL.

LISTA_destruir

Usando la función anterior elimina el contenido de la lista completa, revisando antes si ya estaba vacía.

LISTA_insertar_al_principio

Crea un nuevo nodo y lo inserta al principio de la lista.

LISTA_insertar_al_final

Crea un nuevo nodo y lo coloca al final de la lista.

LISTA_recorrer

Recorre la lista imprimiéndola según el tipo de función que se le agregue para este propósito.

LISTA_destruir_mensaje

Libera la memoria pedida para un mensaje.

LISTA_destruir_usuario

Libera la memoria pedida para la estructura usuario.

LISTA_imprimir_mensaje

Como dice el nombre, imprime un mensaje de la sección de mensajes del usuario.

LISTA_imprimir_usuario_simple

Imprime toda la información del usuario por terminal.

LISTA_imprimir_usuario_multi

Crear un archivo por usuario con la información de estos teniendo de nombre de archivo el numero de id seguido por su nickname.

LISTA_gestion_eliminar

Función usada para la búsqueda y eliminación de usuario según el tipo de búsqueda que fue pedida por terminal.

LISTA_eliminar

Elimina todos los nodos de la lista que se use, liberando la memoria pedida para cada nodo.

LISTA_buscar

Busca en la lista de usuarios el nodo en el que se encuentre el usuario pedido por terminal para su eliminación.

LISTA_cmp_id

Función usada por LISTA_buscar para comparar todos los IDs de la lista de usuarios devolviendo un estado afirmando que lo encontró o no.

LISTA_cmp_usuario

Función similar a LISTA_cmp_usuario con la diferencia de que la comparación la hace entre los nickname de cada usuario.

Problemas encontrados en el desarrollo y sus soluciones

Fugas de memoria

Uno de los grandes problemas que nos encontrados al principio del programa fue las fugas de memoria o memory leaks, el cual se solucionó debuggeando el programa, examinando de forma intensiva al programa para poder liberar toda la memoria dinámica al momento anterior de finalizado el programa.

Orden de los argumentos

Otro de los problemas fue encontrar de que si el usuario escribía como ultima argumento un flag como ejemplo (-e ó --output) entonces el programa fallaba completamente, esto fue rápidamente solucionado colocando una protección para cuando el argumento seguido del flag sea NULL.

Problema para eliminar el primer elemento

Al colocar el nombre o el numero de ID del primer elemento de la lista, provocaba que el programa tomara el argumento como si el usuario se hubiera equivocado en la gramática del argumento, esto se solucionó cambiando la estructura de la función LISTA_gestion_eliminar.

Signo de pregunta en los archivos

Al indicar que la salida multi en los argumentos, se creaban los programas, pero con un signo de interrogación al lado del numero de ID del usuario. Al buscar cual podía ser el fallo vimos que en la sección donde extrayendo el numero de ID al lado tenia un '\n' que la computadora lo estaba interpretando como el signo '?'.

Simulaciones

Falta de argumentos

```
mauricio@mauriciopc ~/9511/git-repos/algo1-tp_final $ ./red_social -e
Error: Falta algunos argumentos para ejecutar el programa
mauricio@mauriciopc ~/9511/git-repos/algo1-tp_final $ ./red_social -o
Error: Falta algunos argumentos para ejecutar el programa
```

```
mauricio@mauriciopc ~/9511/git-repos/algo1-tp_final $ ./red_social -e u:nahuel --output INI
Error: El usuario que quiere eliminar no exista
mauricio@mauriciopc ~/9511/git-repos/algo1-tp_final $ ./red_social -e --output single INI
Error: Los argumentos de eliminar deben ser de la forma u:usuario o i:id
mauricio@mauriciopc ~/9511/git-repos/algo1-tp_final $ ./red_social -e i:23 INI --output
Error: Falta algunos argumentos para ejecutar el programa
```

Incluido de dos usuarios con el mismo ID y/o un ID negativo

```
mauricio@mauriciopc ~/9511/git-repos/algo1-tp_final $ ./red_social -e i:23 --output single INI INI4
No podemos cargar el usuario porque su ID ya exista
No podemo cargar el usuario porque su ID es negativo
No podemo cargar el usuario porque su ID es negativo
[cornelio]
id = 2
nombre = Cornelio Saavedra
amigos = 263,15,23
mensaje = 2069,1811-09-15,2,Hacía falta tanta agua para apagar tanto fueg

[aurelien]
id = 54
nombre = Aurelien Floutard
amigos = 1,33,4,1345
mensaje = 2354190,0000-01-01,54,Hello coucou j'ai hate de vous rencontrer
mensaje = 5678,9999-99-99,99,Bonjour j'espere que vous allez bien
```

Indicaciones de Compilación

Para compilar el programa se tienen que seguir las siguientes pasas (el orden puede variar a excepción del ultimo):

```
$ gcc -ansi -Wall -pedantic -c -o funciones.o funciones.c
```

```
$ gcc -ansi -Wall -pedantic -c -o LISTA.o LISTA.c
```

```
$ gcc -ansi -Wall -pedantic -c -o red_social.o red_social.c
```

```
$ gcc -ansi -Wall -pedantic -o red_social red_social.o LISTA.o funciones.o
```

Para su ejecución se tiene que escribir de la siguiente forma:

```
$ ./red_social -e [busqueda] -o [formato] [[archivo1]...]
```

Aunque se puede escribir en cualquier orden a excepción de que el argumento que siga a flag -e (que también puede ser escrito como --eliminar) tiene que ser si o si [busqueda] y al que le sigue a -o (que puede ser escrito como --output) [formato].

Donde [búsqueda] es el numero id o nombre de usuario de la persona que se busque para eliminar en la red, siguiendo este modelo: i:(número de ID) o u:(nombre de usuario), ejemplo: id:23 o u:cornelio.

Y donde [formato] es la forma en la que quiere mostrar los usuarios de la red, ya sea “simple” con el que se muestran los usuarios por terminal, o “multi” en el que se crea un archivo por cada usuario, conteniendo la información de los mismo, teniendo por nombre de archivo si numero de id seguido por el nickname del usuario en la red social.