TP Final - Programación II

Ariel Leonardo Fideleff

12 de diciembre de 2023

1. Descripción de la solución

1.1. El programa en C

Para la primera parte del TP, se nos pide desarrollar un programa en C que:

1. Tome por argumento el nombre de una persona.
2. Lea un conjunto de archivos `.txt` contenidos en un carpeta cuyo nombre se corresponde con el pasado por argumento. Cada uno contiene textos escritos por la persona en cuestión.
3. Limpiar los textos bajo un conjunto de pautas que permita separarlos en oraciones y evitar que contengan caracteres no deseados.
4. Unificar los textos "sanitizados" en un único archivo bajo el nombre de la persona que los escribió (formato también `.txt`).
5. Llamar al programa en Python con el nombre de la persona como argumento, para continuar a la siguiente etapa de nuestro programa en el TP.

1.1.1. Tomar el nombre de la persona por argumento

Obtener los argumentos que recibe un programa en C es sencillo y no presenta mayores inconvenientes. Más información en detalles de implementación.

Lo único a tener en cuenta es que nos debemos asegurar de que la cantidad de argumentos recibida por la terminal sea la correcta, con el fin de estar seguros de recibir el nombre de la persona necesario para saber en dónde debemos leer los textos a procesar.

1.1.2. Leer los archivos `.txt` escritos por la persona

El enunciado exige el uso de la función `system()` y los comandos `cd` y `ls` para leer los contenidos de un directorio. Esta forma de leer los contenidos de un directorio presenta la desventaja de que no nos permite efectivamente determinar si el directorio deseado, `Textos/[nombre persona]`, exista (o que tengamos los permisos suficientes para leer su contenido).

De hecho, si probamos el comando sugerido en el trabajo directamente en la terminal, observaremos que no se crea el archivo `archivos.txt` si no existe la carpeta, ya que el comando `cd` con el cual intentamos acceder a la carpeta, falla en primera instancia.

El valor de retorno de la función `system()` podría servirnos como indicación de algún error en la ejecución de los comandos... sin embargo observando la documentación se indica que este valor de retorno depende de la implementación de la función, aunque se pueda "esperar" que se corresponda con el código de estado resultante de la ejecución de los comandos[1].

Ante la incertidumbre, optamos por elegir un método con el cual podemos tener mayor seguridad de su correcto funcionamiento. Sabemos bajo haber probado los comandos en la terminal previamente, cómo es el comortamiento de `cd` y `ls` si la carpeta no existe. Entonces, considerando que `archivos.txt` no es creado, podríamos inicialmente comprobar si el archivo existe intentando abrirlo como sabemos hacerlo en C, usando `fopen()`. Al fin y al cabo, si `archivos.txt` no existe,

[1] `https://cplusplus.com/reference/cstdlib/system/`

entonces `fopen` devolvería `NULL`, no? Hay un problema, y radica en que también se nos exige que no eliminemos `archivos.txt` tras haberlo utilizado. En consecuencia, el archivo puede existir como producto de una ejecución anterior correcta, llevándonos a pensar que la carpeta buscada si existe.

Por lo tanto, para resolver esto, antes de ejecutar los comandos asociados al listado de los archivos de la carpeta con `system()`, intentaremos primero sobreescribir `archivos.txt` por uno vacío (o crearlo vacío si no existe). De esta forma, si...

- ... la carpeta deseada **no** existe, al leer `archivos.txt` veremos que se encuentra vacío, y podremos informarle al usuario con un error apropiado deteniendo la ejecución del programa.
- ... la carpeta deseada **sí** existe, el `archivos.txt` sólo resultaría vacío si no se presentan archivos en la carpeta. En este caso, no habría textos de la persona que sanitizar, y no se produciría información para la segunda etapa del programa en Python. Por lo tanto, debemos de informarle de esto al usuario y tampoco queremos que el programa continúe.

Con esto resuelto, si la carpeta existe y hay archivos para procesar, podemos leer el contenido de `archivos.txt` para obtener los *nombres* de los archivos dichos. Como es necesario abirlos para poder procesarlos, debemos primero anteponer la carpeta en la que éstos se encuentran relativa al directorio de ejecución de nuestro programa. Por ejemplo, si la persona se llama `Pepe`, y uno de los textos tiene nombre de archivo `biografia.txt`, su ruta sería `Textos/Pepe/biografia.txt`.

Así, para cada nombre leído, podemos obtener la ruta antes descrita, leer el archivo allí ubicado, y procesar su contenido.

Ahora bien, de esto surge otra decisión importante del programa: es necesario leer los contenidos de los archivos a la memoria para procesarlos? O podemos irlos procesando "directamente" leyendo de los archivos, a la vez que escribimos el contenido sanitizado en el archivo de salida? Es decir, necesariamente el contenido de los archivos debe transitar por la memoria, pero a lo que nos referimos es si es necesario almacenar el contenido de cada archivo en alguna estructura de datos, como un arreglo.

La ventaja de leer primero todo el contenido de los archivos a arreglos, por ejemplo, es que nos permitiría definir funciones que tomen como argumento estas estructuras y que sean independientes de operaciones de lectura / escritura sobre archivos, facilitando la depuración de errores y el testing del procedimiento.

Mientras, como principal desventaja, en C presentamos complicaciones relacionadas a la administración de la memoria. Es decir, deberíamos responsabilizarnos por reservar la memoria suficiente para la información contenida en los archivos, como también de su posterior liberación una vez que no necesitemos usarla más. Esto puede conllevar a un mayor esfuerzo con el fin de evitar errores.

En esta situación particular, podemos analizar que el proceso de sanitización de los archivos no es demasiado complejo y que, desde un punto de vista personal, el esfuerzo requerido para adminstrar la memoria apropiadamente no vale la pena respecto a los beneficios relacionados con la modularidad de las funciones diseñadas. Es más, para una operación como la dada, personalmente creo que si el archivo a procesar es muy grande, sería un "desperdicio de memoria" cargar inicialmente todo el contenido del archivo en la memoria, ya que es un proceso que se puede realizar de forma completamente lineal sobre la entrada.

Por lo tanto, optamos entonces procesar los archivos caracter por caracter, leyendo progresivamente de cada uno de los archivos de la entrada y decidiendo según ciertas condiciones (que consideran las pautas del TP y las decisiones para el paso 3 que se especifican más adelante) qué caracteres deben escribirse al archivo de salida.

Los detalles sobre la implementación se presentarán en el apartado correspondiente.

Como un último comentario, otra de las decisiones tomadas para la lectura de los textos escritos por la persona radica en algo similar a lo anteriormente descrito para los archivos, pero en este caso, sobre `archivos.txt`. Esto vendría a ser, si es preferible leer todos los nombres de archivos contenidos en `archivos.txt` a un arreglo, o si leer progresivamente los nombres, concatenar las rutas, leer y procesar los textos uno por uno, evitando reservar memoria para las cadenas donde se almacenarían las rutas.

En este caso, si bien creo que leer los nombres de los archivos de forma progresiva resultaría en un código más corto, nuevamente esto iría en contra de la modularidad del código. La decisión al fin y al cabo se reduce a preferencia personal, pero pienso que una función que sanitice un conjunto de archivos dados por sus rutas a un único archivo dado por otra ruta, tiene un mejor diseño y modularidad que una función que deba leer un conjunto de nombres de un archivo e irlos concatenando con el directorio en el que se encuentran, para luego irlos procesando y combinando en un archivo de destino.

Es por esto que para la lectura de los nombres de los archivos de la persona a procesar, opté por leerlos en un arreglo para que luego sean procesados por otra función que opere en cada uno, sanitizándolos y escribiendo su contenido limpio en un archivo de salida.

Una vez más, los detalles de implementación se presentarán en el apartado correpondiente.

1.1.3. Limpiar los textos

Para limpiar los textos escritos por la persona, se indican en el enunciado un conjunto de pautas que el archivo combinado resultante debe seguir, junto con un ejemplo de una salida para una entrada dada. Sin embargo, hay ciertos detalles que no son claros, ante los cuales tuve que tomar ciertas decisiones a la hora de programar una función que limpiara un archivo en concreto.

En primer lugar, en teoría se indica que el archivo de salida sólo debe contener saltos de línea y caracteres del alfabeto inglés en minúscula, eliminándose cualquier otro tipo de caracter. Esto no considera los espacios, los cuales son claramente importantes para distinguir las palabras individuales de una oración. Por lo tanto, no queda del todo claro el comportamiento del programa si se presentan múltiples de estos espacios: ¿hay que dejarlos todos? ¿o se debe eliminarlos de forma que las oraciones procesadas se presenten en un formato "normal" bajo las reglas del lenguaje?

Es común el tipo de problemas que plantean "normalizar" una oración, de forma que aquellos espacios que se presenten de forma irregular, sean eliminados. Por lo tanto, decidí tomar esto en cuenta para mi función de sanitización de los archivos. Como un ejemplo, si una oración esta dada por:

```
␣␣␣␣Esta␣es␣␣una␣oracion␣␣␣de␣prueba␣␣.␣
```

La normalizaría a:

```
esta es una oracion de prueba
```

Otro aspecto similar que no está del todo claro, es exactamente a qué se refiere el enunciado con eliminar todos los símbolos. Si bien puede resultar obvio en el uso común de los signos de puntuación, puede ser menos intuitivo si hablamos de "accidentes en la escritura". Por ejemplo, si tenemos:

```
esto_$es una pru3eba, no es as!i?
```

¿Cómo deberíamos limpiar esta oración? Es cierto que el enunciado nunca menciona números, ¿pero cómo haríamos para las palabras `as!i?` o `esto_$es`? Por lo tanto, decidí tomar el criterio general que *cualquier palabra está compuesta por cualquier conjunto consecutivo de caracteres alfabéticos*, concluyendo que cualquier símbolo que se encuentre entre dos conjuntos tales, los divide en dos palabras, que deben estar separadas por un espacio. Esto considera la excepción de que estarían separadas por un salto de línea si alguno de los símbolos entre los conjuntos de caracteres alfabéticos fuera un punto final, indicando el final de una oración. Además, si se presentan muchos símbolos entre dos conjuntos de caracteres alfabéticos, bajo el criterio anteriormente establecido de los múltiples espacios, los conjuntos estarían separados con un único espacio.

Con esto establecido, el ejemplo anterior se limpiaría como:

```
esto es una pru eba no es as i
```

Por último, estableceremos la convención que el archivo de salida debe terminar en un caracter de salto de línea `\n`. Este comentario surge a que algunos de los archivos con textos de personas dados por la cátedra para probar nuestro programa carecían de un `\n` al final de estos archivos. Entonces, si bien un archivo de salida que tenga una oración en cada línea sólo requeriría tener saltos de línea entre oraciones, esto puede generar problemas en algunos casos[2], a lo que elegiremos por incluir un salto de línea al final del archivo siempre que sea posible.

Una excepción a esto sería si se recibe un archivo vacío, en cuyo caso también sería apropiado que se genere un archivo vacío. Esto es especialmente útil si se pretenden combinar muchos archivos en uno, ya que procesar un archivo vacío en un conjunto de archivos no produciría una línea vacía en el archivo final.

1.1.4. Unificar los textos sanitizados

Como indicamos en las decisiones para el paso 2, optamos por procesar los archivos caracter por caracter, escribiendo los caracteres directamente al archivo de salida bajo ciertas condiciones que se van analizando en la entrada.

Entonces, bajo las pautas establecidas en el paso anterior (3), unificar los textos es tan fácil como procesar los archivos en orden, donde el archivo de salida siempre sea el mismo, retomando al procesar cada archivo desde donde se terminó de escribir en la salida para el anterior. Es decir, bajo las decisiones establecidas, es equivalente procesar cada uno de los archivos de entrada a archivos de salida independientes, que procesarlos todos en orden a un mismo archivo de salida, siempre y cuando se escriba el contenido sanitizado de cada archivo a continuación de donde se escribió el del anterior.

Aprovechando que esta sección es relativamente breve, aclararé sobre una decisión tomada en general en relación al manejo de archivos a lo largo del programa en C **y** en el de Python. Tenemos que tener en cuenta que puede darse una situación la cual, por ejemplo, la carpeta `Entradas`, a la cual debemos escribir el archivo de salida, sea inaccesible, sea por no tener los permisos suficientes u algún otro motivo (aunque para el ejemplo debemos asumir por el enunciado que la carpeta existe).

Es por esto que, en general, siempre que intentamos acceder a un archivo para leer o para escribir, optamos verificar además que la operación no nos devuelva una referencia nula (C) o que falle (Python), para poder mostrar en cambio un mensaje al usuario informando sobre la ruta inaccesible.

Más detalles al respecto en el apartado de detalles de implementación.

1.1.5. Llamar al programa en Python

Finalmente, para el último paso debemos llamar al programa en Python con el nombre de la persona como argumento. Esto resulta sencillo siguiendo la indicación del TP de usar una vez más la función `system()`, creando una cadena con el comando que llame al intérprete de Python `python3` con el nombre del programa (se asume que está en el mismo directorio que el ejecutable de C), y pasando como argumento el mismo nombre de la persona que fue recibido como argumento del programa en C.

Los detalles de implementación se encuentran en el apartado correspondiente, aunque no se extienden mucho más de lo indicado en el enunciado.

1.2. El programa en Python

Para la segunda parte del TP, debemos desarrollar un programa en Python que:

1.

[2] `https://stackoverflow.com/questions/729692/why-should-text-files-end-with-a-newline`

2. Detalles de implementación

2.1. El programa en C

2.2. El programa en Python

2.2.1. Tomar el nombre de la persona por argumento

El primer paso es sencillo, ya que la forma más fácil de hacerlo es tomando los argumentos que recibe el programa al ejecutarse por la terminal, con `int argc` y `char *argv[]`, que se pueden usar como argumentos de la función `main`.

2.2.2. Leer los archivos `.txt` escritos por la persona

Como no necesariamente se especifica que se nos brinde el nombre de la persona como argumento de forma correcta tal que se corresponda con uan carpeta existente, o que siquiera se nos provea correctamente un argumento, debemos validar que la cantidad de argumentos recibidos sea la correcta: 2. Uno que siempre está ocupado por el nombre del ejecutable, y el segundo correpondiente al nombre de la persona sin espacios. Si la cantidad de argumentos no es correcta, presentaremos un mensaje con instrucciones de uso del programa al usuario. Entonces, al comienzo de la función `main()` incluimos:

```
// ...
if (argc != 2) {
    puts("usage: ./main [person_name]");
    return 0;
}
// ...
```

2.2.2.1 La existencia de los archivos y los argumentos

Para el segundo paso se presentan decisiones más importantes a tomar, ya que el enunciado nos exige hacer uso de la función `system()` con el fin de obtener una lista de los archivos contenidos en la carpeta donde se encontrarían los textos de la persona. De no presentarse esta exigencia, podríamos haber hecho uso de la librería `dirent.h`, en particular, su función `readdir()` que permite leer el contenido de un directorio[3]. Pero como este no es el caso, debemos adaptar nuestras decisiones a la consigna del trabajo.

Con esto en cuenta, sabemos que debemos de crear un archivo llamado `archivos.txt` que se encontrará en el mismo directorio que el ejecutable de nuestro programa, el cual se creará como resultado de correr el comando `ls` sobre la carpeta en la cual se encontrarían los textos de la persona. Sin embargo, esto puede fallar, ya que podría no existir la carpeta en cuestión. Es cierto que debemos asumir para el trabajo que la carpeta `Textos` debe existir, pero

Por otro lado, habiendo validado que recibimos el nombre de la persona de forma correcta, debemos de verificar que la carpeta que estamos buscando, `Textos/[nombre persona]` exista. Tampoco podemos usar la existencia de `archivos.txt` como referencia, ya que como se nos pide que no se elimine este archivo una vez creado, podría permanecer de una ejecución previa del programa, y contener un listado de archivos que no se corresponde con el argumento provisto en la ejecución actual.

2.3. El programa en Python

3. Correr el programa y tests

4. Documentación (bonus)

[3] `https://stackoverflow.com/questions/4204666/how-to-list-files-in-a-directory-in-a-c-program`