Trabajo Práctico 1 — NombreTP

[7541/9515] Algoritmos y Programación II
Segundo cuatrimestre de 2021

Alumno:	Castillo Pereira, Carlos Eduardo
Número de padrón:	108535
Email:	ccastillo@fi.uba.ar

Índice

1. Introducción

El trabajo práctico presenta el problema de un hospital pokémon en el que se lleva registro de los pokemones y entrenadores ingresados. Estos registros actualmente son escritos a mano en un anotador, sin embargo, se desea informatizar un poco el proceso de almacenando los registros ya existentes en un programa que permita administrarlos con mayor facilidad.

Para facilitar esta tarea, algunos de los archivos que se tenían almacenados fueron pasados a computadora en un formato estilo CSV con la información mínima posible. Cada archivo puede tener una o más líneas, y sigue el mismo formato de campos separados por punto y coma. En cada línea del registro el primer elemento es el ID del entrenador, seguido del nombre del mismo, y a continuación le siguen de a pares, el nombre de cada pokemon, y su respectivo nivel. Por ejemplo:

```
ID_ENTRENADOR1;NOMBRE_ENTRENADOR1;POKEMON1;NIVEL;POKEMON2;NIVEL
ID_ENTRENADOR2;NOMBRE_ENTRENADOR2;POKEMON3;NIVEL;POKEMON4;NIVEL;POKEMON5;NIVEL
ID_ENTRENADOR3;NOMBRE_ENTRENADOR3;POKEMON6;NIVEL
```

Además se tiene constancia de que estos los archivos han sido registrados sin errores. Todos los registros están completos y vienen de a pares. Nunca se va a presentar el caso en el que se encuentre un entrenador sin ID o un pokémon sin nivel.

1.1. Solución propuesta

El principal desafío que plantea este problema es el obtener la información del archivo para luego almacenarla y administrarla de la manera en que se desee. Este proceso se puede dividir en tres tareas fundamentales:

1. Leer el archivo original.
2. Parsear la información obtenida durante la lectura.
3. Estructurar y guardar la información analizada.

Primeramente, se debe obtener la información "cruda" del archivo original. Esto se hace línea por línea, dado que se conoce que cada una de las líneas del archivo contiene datos de un registro (ingreso) diferente. Tras haber obtenido la información no estructurada de cada una de las líneas, esta se debe analizar para identificar cuántos campos vienen escritos en cada línea e interpretar qué representa cada campo.

Finalmente, una vez que se le ha asignado un significado específico a cada campo, se debe organizar la información en el formato correspondiente para poder ser almacenada en el programa y posteriormente utilizada.

2. Detalles de implementación

El programa construido para procesar y almacenar la información de los registros fue escrito en el lenguaje de programación C, siguiendo la especificación del lenguaje detallada en el estándar C99.

Los archivos principales se encuentran dentro del directorio **src** ubicada dentro de la carpeta **archivos** en la raíz del repositorio. Dentro de dicha carpeta se encuentras los siguientes archivos y sus respectivos archivos de cabecera:

- **hospital.c**: Almacena la funcionalidad relacionada con la administración del hospital. Su creación, destrucción, acciones como registrar o liberar un pokémon o un entrenador, entre otros.
- **parser.c**: Almacena la funcionalidad relacionada con el parsing del archivo o el análisis de la información "cruda" obtenida tras la lectura del mismo.

- **split.c**: Incluye la función utilizada para dividir los campos en cada línea durante el parseo del archivo.

Además, el repositorio incluye un archivo **pruebas.c** en el que se encuentran diferentes pruebas automatizadas para detectar errores en la ejecución del programa o en la asignación, manejo y liberación de memoria dinámica. Para esto se utiliza **gcc** como compilador y **valgrind** como herramienta de análisis de memoria.

2.1. Guardado de información

Para proceder con la interpretación de los datos que contiene el archivo, primero debe leerse. El lenguaje de programación C provee bastantes funciones para la lectura y manipulación de archivos con una cantidad y estructura de datos conocida. Sin embargo, en este caso específico, si bien se conoce el formato del archivo y la manera en la que se ordenan los datos, no se conocen ni la cantidad de campos por registro (linea), ni la cantidad total de resgitros, ambos valores pueden variar de archivo en archivo. Para esto se implementó la función **leer_linea**, que utiliza memoria dinámica para poder leer líneas de longitud desconocida.

Una vez se obtiene cada línea del archivo, la misma debe dividirse en campos, denotados con el separador especificado en la descripción del formato de los archivos. Cada campo repesenta un valor diferente que será almacenado según corresponda, por ejemplo, en el caso del primer y segundo campo, se sabe de antemano que estos representan tanto el ID como el nombre de un entrenador, respectivamente, por lo tanto, se utilizarán los valores que almacenen estos campos para definir los valores de las propiedades de los entrenadores que se van a ir guardando por cada nueva línea.

Para facilitar la tarea de registrar un nuevo entrenador a partir de la información obtenida tras la lectura de la línea, se implementó la función **hospital_guardar_entrenador** que reserva el espacio en memoria necesario para guardar un registro de tipo entrenador y posteriormente guarda ese registro creado en el vector que almacena todos los demás registros de entrenadores ingresados al hospital.

```
void
hospital_guardar_entrenador(hospital_t* hospital, entrenador_t* nuevo_entrenador) {
  if (!hospital || !nuevo_entrenador)
    return;

  size_t cantidad_entrenador = hospital_cantidad_entrenadores(hospital);
  entrenador_t* vector_entrenadores_aux = realloc(hospital -> vector_entrenadores,
    (cantidad_entrenador + 1) * sizeof(entrenador_t));
  if (!vector_entrenadores_aux)
    return;

  hospital -> vector_entrenadores = vector_entrenadores_aux;
  hospital -> vector_entrenadores[hospital -> cantidad_entrenador++] = *
    nuevo_entrenador;
}
```

Función 1: Implementación d **hospital_guardar_entrenador**.

Además esta función tiene su homólogo para guardar pokemones en el vector de pokemones del hospital, **hospital_guardar_pokemon**. Ambas funciones son utilizadas al parsear cada línea, funcionalidad encapsulada dentro de la función **hospital_guadar_informacion**.

2.2. Aplicación de función genérica

La función **hospital_a_cada_pokemon** permite al usuario aplicar una función cualquiera sobre todos los pokemones registrados en el hospital, siempre y cuando esta función cumpla con la firma especificada. Dicha función debe ser aplicada a cada pokemon en orden alfabético. Esto implica que el vector que contiene el registro de los pokemones en el hospital debe ser ordenado de manera ascendente en base al nombre de los pokemones.

Sin embargo, si se intenta ordenar el vector de pokemones temporalmente para que esta función genérica sea aplicada, se pierde el orden original que tenían los pokemones cuando fueron ingresados al programa tras la lectura del archivo. A primera vista el perder el orden original de los registros no representa inconveniente alguno al terminar de aplicar la función genérica sobre los pokemones, no obstante, puede ser que eventualmente el usuario quisiera acceder a cierta información sobre los registros y espere que se respete el orden establecido por él al momento de escribir los registros en el archivo original.

Para esto se implementa un algoritmo de ordenamiento (basado en el algoritmo selection sort) que en lugar de ordenar/modificar las posiciones del registro de pokemones que tienen el hospital, se encarga de ordenar un vector de punteros a los pokemones ordenados. Este vector de punteros es creado en base al orden original del vector de pokemones registrados en el hospital, y puede ser ordenado con la función **ordenar_referencia_a_pokemones** sin necesidad de alterar el orden del vector de pokemones guardados en el hospital. Finalmente, se procede a aplicar la función genérica deseada sobre los punteros a cada uno de los pokemones originales de manera ordenada en vez de aplicarla directamente sobre los pokemones guardados sobre en el hospital.

```
void
ordenar_referencia_a_pokemones(pokemon_t** referencias_pokemones, size_t
    cantidad_pokemon) {
  if (!referencias_pokemones || !cantidad_pokemon)
    return;

  for (size_t i = 0; i < (cantidad_pokemon - 1); i++) {
    size_t indice_min = i;
    for (size_t j = i + 1; j < cantidad_pokemon; j++) {
      if (strcmp((*referencias_pokemones[j]).nombre, (*referencias_pokemones[
    indice_min]).nombre) < 0)
        indice_min = j;
    }

    pokemon_t* temp = referencias_pokemones[indice_min];
    referencias_pokemones[indice_min] = referencias_pokemones[i];
    referencias_pokemones[i] = temp;
  }
}
```

Función 2: Implementación de **hospital_a_cada_pokemon**.

```
void
ordenar_referencia_a_pokemones(pokemon_t** referencias_pokemones, size_t
    cantidad_pokemon) {
  if (!referencias_pokemones || !cantidad_pokemon)
    return;

  for (size_t i = 0; i < (cantidad_pokemon - 1); i++) {
    size_t indice_min = i;
    for (size_t j = i + 1; j < cantidad_pokemon; j++) {
      if (strcmp((*referencias_pokemones[j]).nombre, (*referencias_pokemones[
    indice_min]).nombre) < 0)
        indice_min = j;
    }

    pokemon_t* temp = referencias_pokemones[indice_min];
    referencias_pokemones[indice_min] = referencias_pokemones[i];
    referencias_pokemones[i] = temp;
  }
}
```

Función 3: Implementación de **ordenar_referencias_a_pokemones**.

3. Diagramas

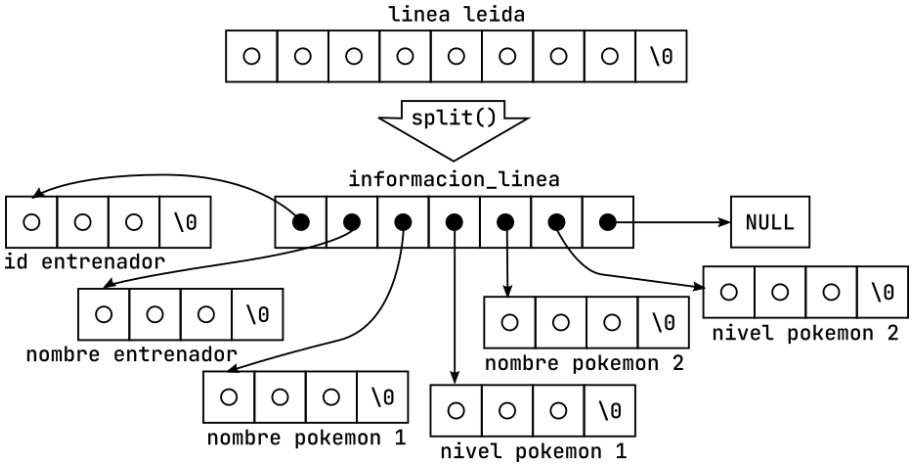

Figura 1: Visualización del split para obtener los campos de cada línea.

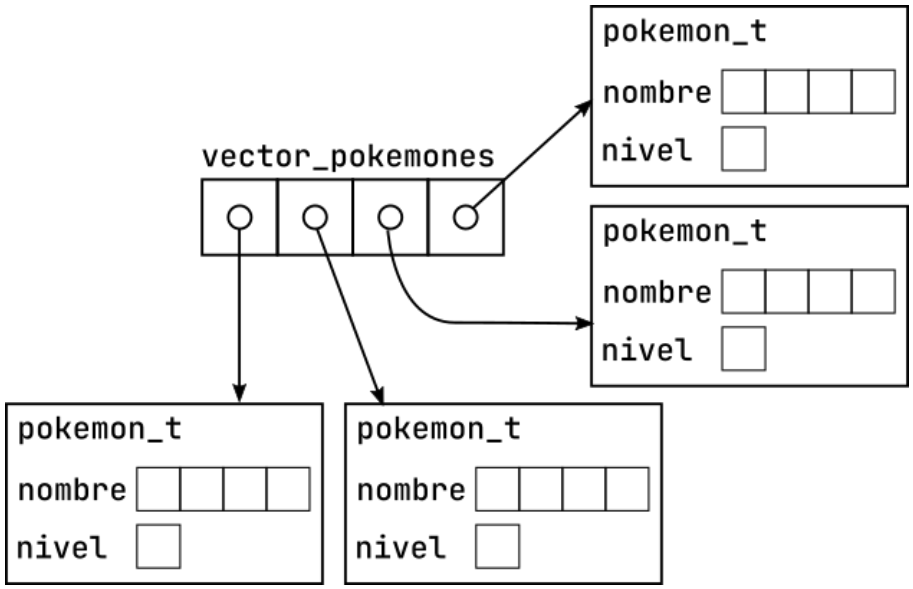

Figura 2: Visualización del vector de pokemones almacenado en el hospital.

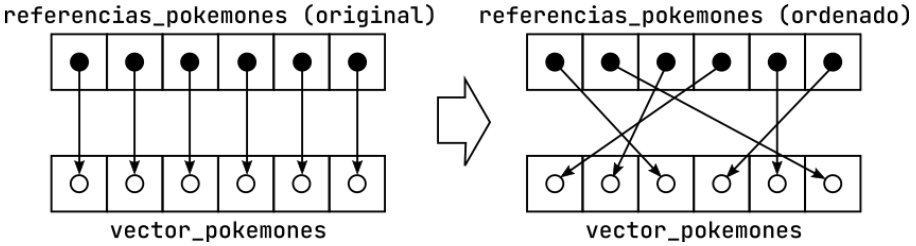

Figura 3: Visualización del vector de referencias a pokemones.