Algoritmos y Programación II
Curso Mendez

.UBAfiuba
FACULTAD DE INGENIERÍA

Salón de la Fama

TP1

Fecha de Presentación: 15/04/2021
Fecha de Vencimiento: 29/04/2021

1. Introducción

El **Hall de la Fama** (Salón de la Fama en Hispanoamérica por el anime, Hall of Fame en inglés) es un salón ubicado en el interior de la Liga Pokémon, en donde se lleva a cabo el **registro de los entrenadores** que han derrotado al Alto Mando y al actual campeón de la liga. Dentro de esta habitación hay una máquina que recupera al equipo Pokémon vencedor y guarda los datos más importantes, tales como el nombre y número de identificación del entrenador y el nombre, especie, género, nivel y zona de captura de cada Pokémon que estuvo presente en el triunfo.

2. Objetivo

El presente trabajo tiene como finalidad que el alumno repase algunos de los conocimientos adquiridos en Algoritmos y Programación I, así como también que comience a familiarizarse con las herramientas a utilizar a lo largo de la materia, tales como el manejo de memoria dinámica y la utilización de bibliotecas.

3. Desarrollo

Para el desarrollo de este trabajo práctico, vamos a usar una metodología de trabajo basado en **pruebas automatizadas**. Las pruebas automatizadas no son mas que fragmentos de código que verifican el correcto funcionamiento de nuestro código. Cada vez que se introduce una modificación a la implementación o se implementa algo nuevo, se deben correr las pruebas automatizadas para asegurarnos de que vamos por un buen camino. Esto nos asegura de que las últimas modificaciones hechas son correctas (según la definicion de nuestras pruebas) y al mismo tiempo nos muestra si los cambios introducidos afectaron de alguna forma la lógica previamente implementada en nuestro sistema.

Esta metodología de trabajo es la que se adoptará también para futuros trabajos (y en general es buena idea adoptarla para el desarrollo de software en general), aunque en esta oportunidad las pruebas seran provistas y solamente será necesario correrlas cuando sea necesario.

Junto con el enunciado se hace entrega de un conjunto de archivos con el esqueleto del trabajo (archivos .h y .c), archivos de salon de prueba y un **makefile** que será de utilidad para compilar y probar el programa. El **makefile** cuenta con las siguientes reglas que pueden ser utilizadas según se necesite:

- compilar: Regla por defecto. Compila el programa principal (main.c).
- valgrind: Corre el programa principal utilizando valgrind.
- test: Corre las pruebas automatizadas con valgrind.

3.1. Parte 1: Memoria dinámica

Para facilitar el desarrollo del tp, aconsejamos empezar implementando algunas primitivas para manejar vectores dinámicos que nos van a servir mas adelante. Estas primitivas tienen como finalidad encapsular las operaciones mas frecuentes que vamos a realizar con vectores dinámicos en **C**. Una vez implementadas, se deben utilizar para el resto del trabajo, evitando duplicar la funcionalidad.

Evitar código duplicado es una parte fundamental del **desarrollo de software** y nos permite tanto reutilizar código como también modificar código de manera mas sencilla y con menos posibilidad de cometer errores. Cualquier modificación al código en cuestión se realiza en un solo lugar, mientras que para código duplicado o triplicado, debemos modificar el código en múltiples oportunidades y lugares diferentes. Esta duplicación de código no sólo aumenta el tiempo de desarrollo si no que también aumenta la cantidad posible de defectos en nuestro código si nos olvidamos de modificar alguna de las copias.

Las primitivas de vectores a implementar son **vtrlen** y **vtradd** (intentando seguir la convención de nombres de las funciones de biblioteca estandar de **C** como **strlen**).

Cuando en **C** hablamos de **strings**, nos referimos a vectores de caracteres delimitados por un carácter **0** literal que marca el final de la cadena de texto.

Por ejemplo el texto "HOLA", sabemos que en memoria se almacena como un vector de 5 elementos: 'H', 'O', 'L', 'A' y '\0'. Intentando extender esta idea, vamos a definir un **vector dinámico** como un vector de elementos de tipo **puntero**,

delimitado por un puntero **NULL** (0 literal) que marca el fin del vector. Entonces, con esta convención, si tenemos un vector de 3 punteros (p1, p2, p3), en memoria se almacena como: p1, p2, p3 y **NULL**.

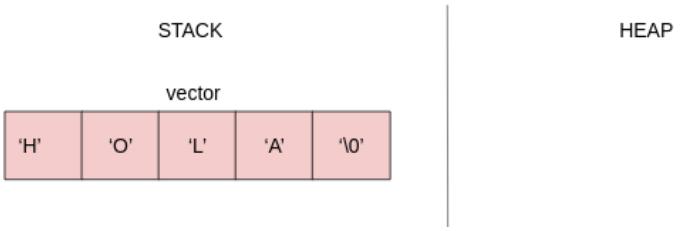

Figura 1: Ejemplo en memoria estatica

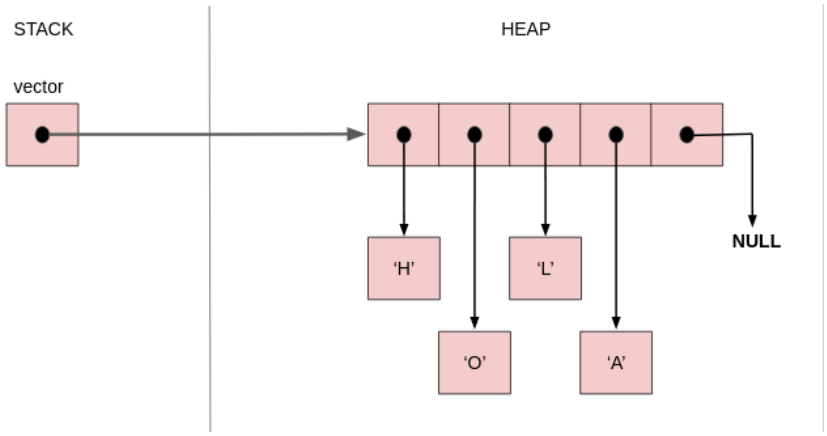

Figura 2: Ejemplo en memoria dinamica

La primera función, **vtrlen** (análoga a **strlen**) recibe un vector dinámico (reservado con **malloc**) y cuenta la cantidad de punteros que contiene dicho vector hasta encontrar el primer puntero **NULL**. Observar que funciona de la misma forma que strlen: recorre los caracteres del vector hasta encontrar el primer elemento 0

La segunda función, **vtradd**, recibe un vector dinámico (reservado con **malloc**) y un puntero y lo que hace es aplicar **realloc** para hacer crecer el vector en un elemento y luego agrega el puntero recibido como último elemento del vector. Recordar que el final del vector se marca con un puntero NULL, por lo que si el vector almacena 4 punteros, en realidad debe tener capacidad para 5 (4 punteros no nulos y uno nulo al final).

Para asegurarse de que la implementación es correcta, se debe correr el comando **make test**. Si la implementación es correcta, los primeros 2 conjuntos de pruebas deberían ser exitosos:

```
➜ implementacion git:(master) ✗ make test
gcc -g -std=c99 -Wall -Werror -o pruebas pruebas.c util.c salon.c
valgrind --leak-check=full --track-origins=yes --show-reachable=yes ./pruebas
==23058== Memcheck, a memory error detector
==23058== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==23058== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==23058== Command: ./pruebas
==23058==

Pruebas de vectores
===================

vtrlen:
=======
✓ vtrlen devuelve la cantidad de punteros no nulos de un vector de punteros.
✓ vtrlen de un vector nulo es 0.

vtradd:
=======
✓ vtradd de un vector nulo resulta en un nuevo vector.
✓ el nuevo vector tiene longitud 1.
✓ vtradd de otro elemento en el vector resulta en un vector no nulo.
✓ el nuevo vector tiene longitud 2.
✓ vtradd de otro elemento en el vector resulta en un vector no nulo.
✓ el nuevo vector tiene longitud 3.
✓ El primer elemento del vector es el primer elemento agregado.
✓ El segundo elemento del vector es el segundo elemento agregado.
✓ El tercer elemento del vector es el tercer elemento agregado.
```

Figura 3: Salida esperada

3.2. Parte 2: Parseo de texto

Este trabajo consiste en leer archivos de texto estructurado separado por **;** que deben ser leidos e interpretados correctamente para poder luego crear **entrenadores** y **pokemon** según corresponda. Cada línea del archivo contiene o bien los datos de un **entrenador** o bien los de un **pokemon**. Siempre un **entrenador** debe aparecer primero y todos los **pokemon** que aparezcan a continuación perteneceran a dicho **entrenador**. No se evalúa en esta oportunidad la lectura de archivos mal formados y se asume que siempre se va a encontrar primero un **entrenador** en el archivo.

El formato del archivo de texto es el siguiente:

```
nombre_entrenador1;victorias
nombre_pokemon1;nivel;fuerza;inteligencia;defensa;velocidad
nombre_pokemon2;nivel;fuerza;inteligencia;defensa;velocidad
nombre_entrenador2;victorias
nombre_pokemon3;nivel;fuerza;inteligencia;defensa;velocidad
```

Figura 4: Formato del archivo

En este ejemplo se muestra un archivo con 2 entrenadores (nombre_entrenador1 y nombre_entrenador2) con las victorias que posee cada uno en la liga. El primer entrenador posee 2 pokemon mientras que el segundo posee 1 solo. Notar que cada línea está formada por diferentes campos (que se corresponden con los campos de las estructuras de datos) separadas por **;**.

Para facilitar la interpretación de estas líneas, se va a implementar la función **split**. Esta función recibe una línea de texto (char*) y un caracter separador (en este caso ;) y devuelve un vector dinámico que contiene en cada posición uno de los substrings delimitados por el caracter dado. Por ejemplo, para el caso de la segunda línea de texto, el resultado es un vector de 6 punteros, donde el primer puntero apunta al string 'nombre_pokemon1', el segundo apunta al string 'nivel', el tercero al string 'fuerza', etc.

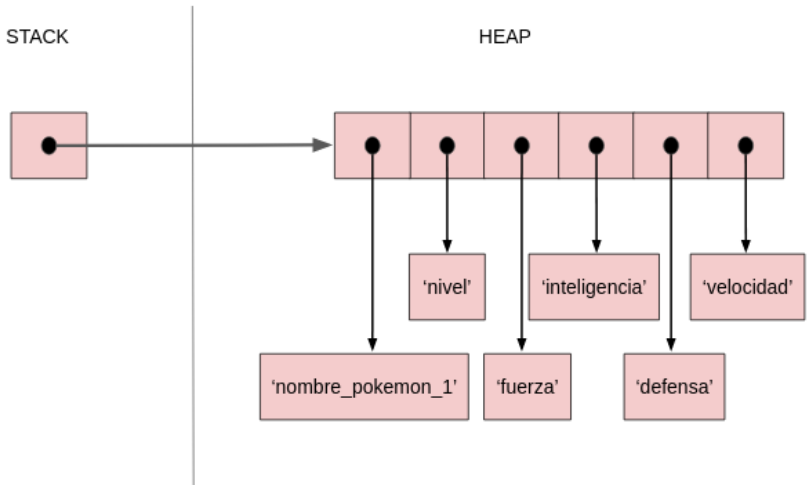

Figura 5: Ejemplo mencionado

Utilizando esta función **split** luego podemos leer el archivo línea a línea, separar el texto delimitado por **;** y obtener un vector de los campos de cada línea. Esto nos permite identificar si se trata de un entrenador (2 campos) o de un pokemon (6 campos), y posteriormente nos permite parsear cada uno de los campos según sea necesario (por ejemplo utilizando **atoi** o **strtol** para convertir los campos numéricos de strings a números).

Recordar correr las pruebas con **make test** para verificar que la solución es la correcta.

```
split:
======
✓ split sobre un texto no vacío resulta en un vector de elementos
✓ Se obtuvo un vector de 3 elementos
✓ El primer elemento es Pikachu
✓ El segundo elemento es 999
✓ El tercer elemento es 1
✓ split sobre un texto sin separador devuelve un vector
✓ Se obtuvo un vector de 1 elemento
✓ El unico elemento del vector es el string completo
✓ split sobre un texto con campos vacíos devuelve un vector
✓ Se obtuvo un vector de 4 elementos
✓ El primer elemento del vector es el primer elemento del string
✓ El segundo elemento del vector es un elemento vacío
✓ El tercer elemento del vector es un el tercer elemento del string
✓ El cuarto elemento del vector es un elemento vacío
✓ Split de un string vacío es null
✓ Split de un string null es null
```

Figura 6: Salida esperada

3.3. Parte 3: Lectura de archivos

Leer archivos de texto estructurados es una actividad que debería ser trivial pero que presenta sus dificultades si uno se encuentra utilizando solamente la biblioteca estándar de **C99**. No existe ninguna función que nos permita leer una línea de texto completa, por lo cual vamos a implementar una propia.

La función que más se acerca a lo que queremos hacer es ***fgets***. La función ***fgets*** recibe un archivo y un buffer de memoria y lee una línea completa de texto (hasta encontrar un '\n', un **enter**) o hasta que se acabe el buffer de memoria provisto por el usuario. Obviamente que el usuario no puede conocer de antemano el tamaño total de la línea de texto de un archivo, por lo cual es necesario realizar múltiples llamadas a ***fgets*** hasta poder leer toda la línea.

Proponemos entonces que se implemente la función ***fgets_alloc***, que recibe un archivo abierto y lee una línea de texto (hasta encontrar '\n') y devuelve un vector dinámico reservado en el heap con el texto leído. Para implementarla sugerimos reservar un vector de tamaño inicial 512 bytes y luego ir realizando reallocs agrandando el buffer de a 512 bytes cada vez.

Nuevamente se puede verificar el correcto funcionamiento de la implementación corriendo el comando **make test**. Las pruebas correspondientes a la categoría de lectura de archivos debería salir en verde.

```
Pruebas de archivos
===================

fgets_alloc
===========
✓ Puedo leer una línea completa de un archivo de texto.
✓ La línea se leyó completa
✓ No puedo leer una segunda línea.
✓ Puedo leer la primer línea de un archivo de 3 lineas.
✓ La línea se leyó completa
✓ Puedo leer la segunda línea.
✓ La línea se leyó completa
✓ Puedo leer la tercer línea.
✓ La última línea se leyó completa
✓ Puedo leer una línea de un archivo con una línea larga de texto
✓ La línea se leyó completa
```

Figura 7: Salida esperada

3.4. Parte 4: API

La **API** o **Application Programming Interface**, es la interfaz que separa nuestra implementación del 'resto de mundo'. En este caso, nuestro trabajo implementa funciones que en conjunto permiten leer y escribir archivos del salón de la fama pokémon. La interfaz permite que alguien que quiera leer y escribir archivos del salón de la fama no tenga que conocer todos los detalles asociados a dichos archivos. Basta con invocar alguna de nuestras funciones para que pueda realizar la operación en cuestión. Esta interfaz es entonces el punto de contacto entre nuestra implementación y el programador que la va a utilizar.

Muchas veces se suele hablar de un contrato cuando hablamos de interfaces. Esto significa que nuestra interfaz se

compromete a hacer una cosa específica dados ciertos parámetros de entrada. Si quien utiliza nuestra interfaz cumple su parte de proveer los datos adecuados, nuestra implementación debe cumplir su parte del contrato realizando las operaciones que dicen implementar.

Por ejemplo: Si decimos que nuestra función **salon_leer_archivo** recibe un nombre de archivo de salón válido y devuelve un salón en memoria con los datos del archivo, si un usuario provea un nombre de archivo válido, es justo que nosotros (dentro de las posibilidades) le devolvamos el salón con los datos cargados en memoria. El no cumplimiento del contrato por la implementación no es un comportamiento aceptable.

En esta parte del trabajo vamos a implementar las funciones que forman parte de nuestra interfaz (el archivo **salon.h**) y vamos a asegurarnos de que su funcionamiento sea el esperado según la descripción de cada una.

Es importante resaltar que el comando **make test** es fundamental a la hora de implementar cada una de las funciones. Uno de los errores más comunes es el de implementar en bloque varias funciones de la interfaz para luego intentar compilarla. Esto suele llevar a un mundo de dolor y sufrimiento (re exagerado) y por lo tanto se aconseja correr el comando **make test** frecuentemente durante el desarrollo para asegurarnos del correcto funcionamiento de cada una de las partes del trabajo. Si todo funciona correctamente, la cantidad de pruebas fallidas debería ser **0** y **valgrind** debe mostrar que se encontraron 0 errores durante la ejecución de las pruebas.

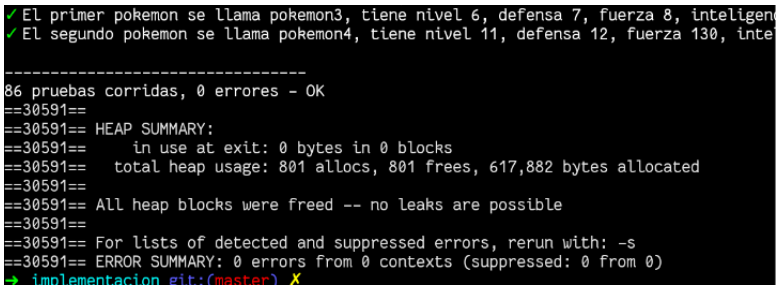

Figura 8: Salida esperada

3.5. Parte 5: Aplicación principal

Por último vamos a implementar una función principal (en un archivo **main.c**) que simplemente haga uso de nuestra API salón y que debe realizar las siguientes operaciones:

- Crear un salón usando el archivo 'salon_original.sal'
- Obtener los entrenadores con al menos 3 ligas ganadas y mostrarlos por pantalla.
- Agregar 2 entrenadores al salon con 5 y 7 victorias respectivamente.
- Obtener los entrenadores con al menos 5 ligas ganadas y mostrarlos por pantalla.
- Guardar el salón a un nuevo archivo 'salon_modificado.sal'
- Salir con valor de retorno 0

En caso de error en cualquier momento se debe salir con valor de retorno 1

4. Bibliotecas propuestas

4.1. Estructuras

```c
#define MAX_NOMBRE_POKEMON 10
#define MAX_NOMBRE_ENTRENADOR 30

typedef struct{
        char nombre[MAX_NOMBRE_POKEMON];
        int nivel;
        int fuerza;
        int inteligencia;
        int velocidad;
```

```
        int defensa;
}pokemon_t;

typedef struct{
        char nombre[MAX_NOMBRE_ENTRENADOR];
        int victorias;
        pokemon_t** equipo;
}entrenador_t;

typedef struct{
        entrenador_t** entrenadores;
}salon_t;
```

4.2. API pública

```
/*
 * Lee archivo y lo carga en memoria.
 *
 * Si no puede leer el archivo o hay un error, devuelve NULL.
 */
salon_t* salon_leer_archivo(const char* nombre_archivo);

/*
 * Guarda el salon a un archivo.
 *
 * Devuelve la cantidad de entrenadores guardados o -1 en caso de error.
 */
int salon_guardar_archivo(salon_t* salon, const char* nombre_archivo);

/*
 * Agrega un entrenador al salon.
 *
 * El entrenador, como todos los pokemon del mismo, deben residir en memoria
 * dinamica y debe ser posible de liberar todo usando free. Una vez agregado al
 * salon, el salon toma posesion de dicha memoria y pasa a ser su responsabilidad.
 *
 * Devuelve el salon o NULL en caso de error.
 */
salon_t* salon_agregar_entrenador(salon_t* salon, entrenador_t* entrenador);

/*
 * Busca en el salon entrenadores que hayan ganado por lo menos
 * cantidad_minima_victorias batallas.
 *
 * Devuelve un vector de entrenadores a liberar por el usuario usando free.
 */
entrenador_t** salon_obtener_entrenadores_mas_ganadores(salon_t* salon, int
cantidad_minima_victorias);

/*
 * Muestra por pantalla los datos de un entrenador y sus pokemon.
 */
void salor_mostrar_entrenador(entrenador_t* entrenador);

/*
 * Libera toda la memoria utilizada por el salon, los entrenadores y todos los
 * pokemon
 */
void salon_destruir(salon_t* salon);
```

4.3. utiles.h

```
#ifndef UTIL_H
#define UTIL_H

#include <stdlib.h>
#include <stdio.h>

/*
 * Dado un vector de punteros (finalizado en NULL), devuelve la cantidad de
 * punteros en el vector (sin contar el nulo al final)
 */
```

```c
size_t vtrlen(voir** ptr);

/*
 * Agrega un item a un vector de punteros dinamico.
 *
 * Si ptr es NULL, se crea un nuevo vector.
 * Devuelve un puntero al nuevo vector de punteros.
 */
void* vtradd(void* ptr, void* item);

/*
 * Aplica la funcion free al vector de punteros y a todos los
 * punteros contenidos en el mismo
 */
void vtrfree(void* ptr);

/*
 * Divide un string cada vez que encuentra el separador dado y
 * devuelve un vector de strings
 */
char** split(const char* str, char separador);

/*
 * Lee una linea completa de un archivo y devuelve un puntero al string leido.
 * El string devuelto debe ser liberado con malloc.
 */
char* fgets_alloc(FILE* archivo);

/*
 * Si el archivo no es nulo, lo cierra con fclose.
 */
void fclosen(FILE* archivo);

#endif /* UTIL_H */
```

5. Entrega

La entrega deberá contar con todos los archivos necesarios para compilar y ejecutar correctamente el TP (incluídos los archivos de prueba y el **makefile** original).

Dichos archivos deberán formar parte de un único archivo **.zip** el cual será entregado a través de la plataforma de corrección automática **Chanutron2021** (patente pendiente).

El archivo comprimido deberá contar además con:

- Un **README.txt** que contenga:
  - Una breve explicación de la solución imlplementada,
  - Cualquier aclaración que necesite hacer acerca de la implementación realizada (decisiones de diseño) y que su corrector necesite saber al corregir su trabajo.
  - Un breve desarrollo de los siguientes temas teóricos:
    1. Heap y stack: ¿Para qué y cómo se utiliza cada uno?
    2. Malloc/Realloc/Free: ¿Cómo y para qué se utilizan?
- Este enunciado.

6. Referencias

- https://pokemon.fandom.com/es/wiki/Hall_de_la_Fama