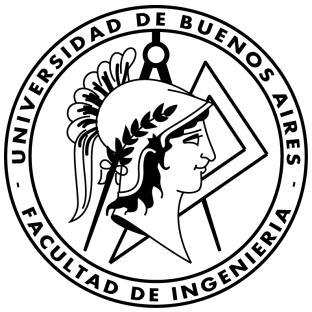

Trabajo Práctico 1 — Cajas Pokemon

Algoritmos y Programación II
Segundo cuatrimestre de 2022

Alumno:	Fernández, Facundo Nahuel
Número de padrón:	109097
Email:	ffernandez@fi.uba.ar

Índice

1. Introducción

El objetivo de este primer TP es implementar la funcionalidad del Sistema de almacenamiento Pokemon (se basa en cajas que contengan a los pokemones, ordenados alfabéticamente dentro de ellas) con con los requerimientos especificados en el enunciado y en el propio archivo .h. Básicamente, se pide leer/escribir pokemones desde/hacia archivos .csv; almacenar los pokemones leídos en memoria e implementar funciones que permitan consultar los datos leídos. Los archivos csv contienen una o mas líneas con información de los Pokemon. La información presente es: Nombre (máximo 20 caracteres), nivel (entero), poder de ataque (entero) y poder de defensa (entero).

2. Teoría

1. ¿Qué son y cómo funcionan los punteros?

   Un puntero es una variable que almacena una dirección de memoria en donde habrá un tipo de dato, según el tipo de dato al que apunte dicho puntero.

   Existen dos tipos de operadores:

   1. De dirección, "&", que devuelve la direccion del operando. (ver ejemplo en Figura 1, de ese modo, el puntero definido tendrá guardada la dirección de memoria donde se encuentra 'entero').

   2. De indirección "*", que devuelve el valor del objeto hacia el cual su operando apunta (debe ser un puntero). Si queremos imprimir por pantalla el valor de *ptr_entero, se imprimirá el número 2, pues es el valor de la variable a la que apunta este puntero.

   Un arreglo es un tipo de dato estructurado en el cual los elementos del mismo se encuentran un después del otro en la memoria. Por ejemplo, si defino un vector como se muestra en Figura 2 (se muestra también cómo acceder a cada posición, de dos maneras equivalentes), tendré reservados en el stack diez espacios en los que entre un entero. El problema reside en que no podré modificar el tamaño de este vector nunca durante la ejecución, por lo que necesitaré utilizar memoria dinámica, es decir, una forma de manejar el heap para reservar y liberar la memoria que necesite según cada caso. Para esto servirán malloc, realloc y free.

2. Funcionamiento y correcto uso de malloc y realloc.

   La función malloc reserva bytes (tantos como se le pase por parámetro) y devuelve un puntero a la memoria reservada.

   Ejemplo: int puntero_entero = malloc(sizeof(int)); –>Reserva la cantidad de bytes de memoriad devuelta por sizeof(int), que serán cuatro bytes, una vez que se hizo esto, se asigna a la variable puntero_entero la dirección de memoria que malloc reservó en el heap.

   Para utilizar memoria dinámica, se reserva utilizando malloc y se libera usando free. Si malloc no pudo reservar memoria, asignará al puntero el valor NULL, por lo que se deberá verificar si esto sucede y, en ese caso, interrumpir el programa para que no haya problemas.

   La función free libera espacio de memoria apuntado por un puntero e inicializa a este puntero con NULL.

   Ejemplo: free(puntero_entero); —>Libera los cuatro bytes de memoria dinámica en el heap que había sido reservada con malloc y luego inicializa a puntero_entero con NULL.

   Ahora bien, si quiero agrandar el tamaño de un bloque en tiempo de ejecución porque el espacio que había reservado en un principio no es suficiente, debo reservar un nuevo bloque más grande, liberar el anterior y asignarle al puntero que guardaba el bloque original la nueva dirección de memoria que quiero reservar.

   De esto se encargará realloc: es una función que recibe el puntero que almacena una dirección de memoria y un size_t y modifica el tamaño del bloque de memoria al que apunta dicho

puntero por el tamaño que se indique (la variable de tipo size_t que se le mande por parámetro), reservando un nuevo bloque de memoria y liberando el antes ocupado. Si el nuevo tamaño es mayor que el anterior, la memoria añadida no estará inicializada. Si el puntero es NULL, esto equivale a malloc(size).

El realloc siempre se debe hacer sobre una variable auxiliar. Si se produce un error al intentar reservar memoria, se debe interrumpir el programa. Si no, se guarda en el puntero en el que estaba guardada la nueva dirección de memoria reservada por realloc.

3. Detalles de implementación

Se explicará el motivo de cómo definí las estructuras, tanto la de los pokemones como las de las cajas.

Para los pokemones, la estructura está formada por cuatro variables, una de tipo char[] llamada nombre y tres de tipo int: nivel, poder ataque y poder defensa. Los nombres de cada variable son bastante declarativos, cada pokemon tiene un nombre, un nivel y dos tipos de poderes, que serán guardados en dichas variables.

Las cajas fueron pensadas como vectores de punteros a pokemones, por lo que la estructura de tipo caja-t está formada por dos variables: una de tipo pokemon_t** (pokemones) y otra de tipo int (cantidad). La primera guardará los punteros a pokemones y la segunda la cantidad de pokemones que hay en la caja. ¿Por qué cada caja es un vector de punteros a pokemones y no un vector de pokemones? La biblioteca cajas.h tiene incluida la biblioteca pokemon.h, en la que es definida una estructura del tipo pokemon_t. Solo podré acceder a las variables de esta estructura desde pokemon.c y NUNCA desde cajas.c (es una estructura incompleta aquí, no sé el tamaño de pokemon-t ni las variables que conforman la estructura). Lo que sí es posible es utilizar funciones de pokemon.h (que reciben punteros a pokemones) desde cajas.c, por lo que podré "manipularçada dirección de memoria en el vector pokemones con dichas funciones.

La forma de compilar el tp es acceder desde la terminal donde esté ubicada la carpeta enviada (que debe contener los archivos csv enviados) y compilar (debe tenerse instalado gcc) ingresando en la terminal 'gcc main.c src/*.c -o ejecutable' (y los flags de compilación que se deseesn) y debe correrse ingresando './ejecutable'.

La idea de main.c es que se lean dos archivos y se guarden en dos cajas diferentes, imprimiendo el contenido de ambas cajas en la terminal. Además, se genera una tercera caja con caja-combinar, de la cual se mostrarán los pokemones en ella en un archivo de salida llamado pokemones3.csv (asegurarse de que no haya un archivo con el mismo nombre dentro de la carpeta o esté será reemplazado por el nuevo).

3.1. Detalles de pokemon crear desde string

Se crean cuatro variables locales (nombre de tipo char[], nivel, ataque y defensa de tipo int) en las que se guarda el contenido que se lee en el string (el cual se verifica que sea válido anteriormente) que recibe la funcion. Si la cantidad de variables que se guarda es correcta (cuatro), recién en ese momento reservo memoria para crear un puntero a un pokemon, y en cada variable del pokemon guardo los valores de las cuatro variables locales que se guardaron leyendo el string.

3.2. Detalles de caja cargar archivo

Se contemplaron los casos en los que se recibe un archivo con lineas mal escritas o archivos vacíos (no estaban considerados en las pruebas de chanutron´t). En ambos casos, se devuelve NULL y no se crea ninguna caja (teniendo en cuenta que en la descripción presente en cajas.h dice "Si el archivo no existe, ES INVALIDO o no se puede reservar memoria, se debe devolver NULL."). Después de todo caso de error, se libera el espacio de memoria que se había reservado para la caja y los pokemones dentro de ella (puede darse el caso de un archivo que tenga líneas

bien escritas pero luego se encuentre una que no tiene formato válido) y se cierra el archivo abierto para leer.

Si el archivo que se lee es válido, los pokemones se van guardando uno a uno (aumentando la memoria reservada para la caja cada vez que se agrega un pokemon) y se los ordena una vez almacenados correctamente. Ver diagrama x (ejemplo, tres pokemones guardados y se reservo espacio para guardar un cuarto): cuando se reserva memoria para guardar un puntero a pokemon, hasta que no guardo el pokemon al que quiero apuntar no sé que hay en ese lugar de memoria que guarde. (Ver figura 3)

4. Diagramas

```
1   int un_entero =2;
2   int * ptr_entero ; // un puntero a un entero
3
4   ptr_entero = &un_entero ;
```

Figura 1: Ejemplo puntero.

```
1   int v[10];
```

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]
*(v)	*(v+1)	*(v+2)	*(v+3)	*(v+4)	*(v+5)	*(v+6)	*(v+7)	*(v+8)	*(v+9)

Figura 2: Ejemplo vector.

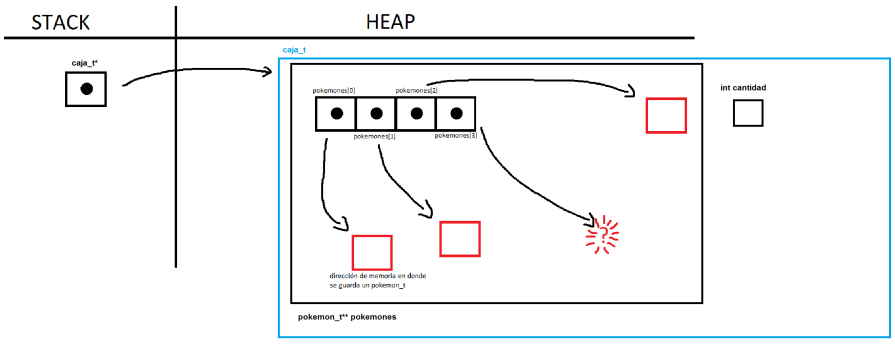

Figura 3: Caja con tres pokemones guardados y un cuarto lugar de memoria reservado (sin guardar pokemon).