INFORME TP 1 - Romina Aylen Gimenez - 107868.

Uso de MAKEFILE:

compilar:
gcc main.c src/pokemon.c src/cajas.c -o tp1 -g -std=c99 -Wall -Wconversion -Wtype-limits -pedantic -Werror -O2

valgrind:
sudo valgrind --leak-check=full --track-origins=yes --show-reachable=yes --show-leak-kinds=all ./tp1

ejecutar:
sudo ./tp1

Aclaración: uso “sudo” porque por alguna razón, en valgrind, no me dejaba crear el archivo de salida sin usar los privilegios.

Un **puntero** es una variable que representa la posición de otro dato, tal como una variable o un elemento de un array.
Se diferencian dos tipos de punteros, que son los **punteros a datos** y los **punteros a funciones**, no podemos castear un puntero a dato para que sea un puntero a función, ni viceversa.
Un **puntero a función** es una variable que almacena la dirección de memoria de una función. Al igual que un **puntero a dato** es una variable que almacena la dirección de memoria de un dato que está en memoria.

Un registro o struct en C es un tipo de **dato estructurado**, que define a una **lista de variables agrupadas físicamente** bajo un mismo nombre en un bloque de memoria, permitiendo de esta forma que estas sean accedidas mediante la utilización de un único nombre.
Las variables declaradas dentro de las llaves de la definición de estructura son los miembros de la estructura. Esto permite contener, ya sea a otros tipos de datos estructurados como a tipos de datos simples.
Las estructuras pueden ser pasadas a funciones, pasando miembros de estructura individuales o pasando toda la estructura.
Para pasar una estructura en llamada por referencia tenemos que colocar el ’*’ o ’&’, dependiendo el caso. Los arreglos de estructura como todos los

demás arreglos **son automáticamente pasados en llamadas por referencia**. Si quisiéramos pasar un arreglo en llamada por valor, podemos definir una estructura con único miembro el array. Una función puede devolver una estructura como valor.

Dentro del TP hay dos estructuras defindas:

```
pokemon_t {
        char nombre [MAX_NOMBRE_POKEMON];
        int nivel;
        int poder_ataque;
        int poder_defensa;
        }

cajas_t{
        pokemon_t ** pokemones_guardados;
        int cantidad_pokemones_guardados;
}
```

cajas_t guarda un puntero a un vector dinámico de punteros de pokemon_t, el cual se expande a medida que crean pokemones, con los datos de la línea leída del archivo.
La implementación anterior obliga a primero crear la caja, después crear el vector dinámico y finalmente , cada uno de los pokémones con cuya dirección de memoria, se rellenan en el vector.
Ejemplo para acceder al nombre del pokemon numero n:

```
caja_t ->pokemones_Guardados[n]->nombre
```

Por último, esta implementación, también obliga a liberar la memoria de todos los pokemon primero, después, a liberar el vector dinámico que los contiene y finalmente la caja que contuvo al vector, para evitar pérdida de memoria,

Los variables y vectores en C ocupan un tamaño prefijado, **no pueden variar durante la ejecución** del programa.
Por esta razón, es que existen varias funciones estándares de la biblioteca <stdlib.h> que permiten por medio de **punteros** reservar o liberar **memoria dinámica en tiempo de ejecución** en el Heap del programa.
Para utilizar este tipo de memoria el programador debe hacerlo de forma explícita solicitándolo al sistema operativo. Para ello en el lenguaje de programación C se utilizan funciones de la biblioteca

La función **malloc** (abreviatura del inglés ***memory allocation***) sirve para solicitarle al sistema operativo , que se desea utilizar memoria del heap, y **devuelve** un puntero a la memoria reservada. Malloc devuelve un **puntero void** al espacio asignado o NULL **si no hay suficiente memoria disponible**.

La función **realloc** (abreviatura del inglés ***memory reallocation***) sirve para **modificar el tamaño del bloque de memoria apuntado** por puntero en size bytes. El contenido del bloque de memoria permanecerá sin cambios desde el inicio del mismo hasta el mínimo entre el viejo y nuevo tamaño. Si el nuevo tamaño del bloque es mayor que el tamaño anterior, la memoria añadida no se encuentra inicializada en ningún valor. Si el puntero es NULL, entonces la llamada es equivalente a malloc(size) para cualquier valor de size.
Sí el size es cero y el puntero no es NULL entonces la llamada es equivalente a free(puntero).

La función **free() libera espacio de memoria reservada**, apuntado por el puntero que debe haber sido devuelto previamente por una llamada a la función malloc(), calloc() o realloc(). De otra forma, si el free(puntero) ha sido ya ejecutado anteriormente, ocurrirá un comportamiento no definido.
Si el puntero es NULL, la operación no se realiza.