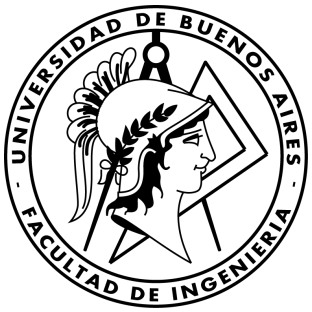

Trabajo Práctico n° 1 - Cajas Pokemon

[7541/9515] Algoritmos y Programación II
Primer cuatrimestre de 2022

Alumno:	Bohórquez, Rubén
Número de padrón:	109442
Email:	rbohorquez@fi.uba.ar

Índice

1. Introducción

Este TP se centró en trabajar con memoria dinámica. La cosigna era implementar las estructuras y las funciones presentes en dos archivos: **pokemon.h** y **cajas.h**. Las funciones presentes en el primero tenían que ver con crear, leer y destruir pokemones; y las del segundo con crear, leer y destruir cajas de pokemones, trabajando con archivos csv. Para este trabajo se necesitaba trabajar con vectores de tamaño dinámico, por lo que un buen uso de malloc(), realloc() y free() era necesario. A continuación se presentan los fundamentos teóricos para el uso de memoria dinámica en C, seguido de un par de precisiones sobre la implementación de las funciones pedidas.

2. Parte teórica - memoria dinámica y punteros

La parte central de este trabajo fue trabajar con memoria dinámica para construir, entre otras cosas, vectores de tamaño variable. A continuación se presenta una breve introducción a los conceptos que se trabajaron:

2.1. La memoria - heap y stack

La memoria es el lugar donde la computadora almacena los datos con los que está trabajando. Del mismo modo que, cuando uno se sienta a estudiar, pone en la mesa todas las cosas que necesita (libros, cuaderno, lapiceras...), un programa cuando se ejecuta pone en memoria los datos que necesita y que va generando (variables, constantes, el código del programa en sí, etc.). Se puede pensar la memoria como una fila muy larga de casilleros, donde en cada cajita hay guardado un valor. El programa que estés usando luego interpreta el valor de una (o varias) de esas cajitas como un número, una letra, un pokemon, etc. Cada casillero tiene también una etiqueta, así la computadora puede recordar dónde guardo cosas en la memoria. Estos casilleros son de 1 byte (8 bits) de tamaño, y un dato puede ocupar varios de estos casilleros.

La memoria está dividida en varios espacios diferentes, pero a nosotros nos interesan dos: el stack y el heap. Ahí, el programa almacena variables que está utilizando.

1. **El stack**, similar a las otras secciones de memoria del programa (code y data), no puede ser controlado directamente por el programador. El compilador (programa que convierte el código fuente que puede leer un humano en código que puede leer la computadora) y el sistema operativo se encargan de reservar la memoria del stack. Ahí, el programa guarda las variables de las funciones sobre las que está trabajando. Un número, un caracter, un pokemon, o una lista de tamaño fijo son todas cosas que vivirían en el stack de un programa. El sistema operativo se encarga de asignar un espacio en la memoria del dispositivo al stack cuando se lanza el programa.

2. **El heap**, que es la sección de la memoria dinámica del programa. El programador tiene control sobre esta parte de la memoria, por lo que puede escribir código que la manipule mientras se ejecuta el programa. Por eso, la memoria del heap no se reserva cuando se empieza a ejecutar el programa, sino que se va reservando y asignando según haga falta durante la ejecución. Es responsabilidad entonces del programador hacerse responsable de asignar y liberar toda la memoria del heap que vaya a usar. Si bien los sistemas operativos modernos saben como lidiar con programas que hacen mal uso de la memoria, máquinas más simples (como sistemas embebidos) que también necesiten correr ese programa pueden no saber como, por ejemplo, liberar memoria que un programa reservo y nunca marcó como libre.

2.2. Punteros - trabajando con memoria dinámica en C

La manera en la que se trabaja con el heap en C es con la utilización de **punteros**. Un puntero es un tipo de dato que almacena una una dirección a memoria. Así como un *int* representa un

número entero, y un *char* un caracter ASCII, un ***int**** representa un *puntero a* un entero (la dirección de memoria en la que vive un entero). Los punteros pueden apuntar a una variable en el stack o en el heap, e incluso pueden apuntar a otros punteros:

```
int numero = 27; \\Esto es un numero normal
int *ptr_a_numero = &numero; \\Guarda la direccion de memoria del numero
int **ptr_a_ptr = &ptr_a_numero; \\Guarda la direccion de memoria del puntero
```

Como se puede ver, si yo tengo una variable *var* del tipo *tipo*, la dirección de memoria de esa variable se puede obtener con *&var*, y un puntero a esa variable sería del tipo *tipo**. También se puede acceder a los valores guardados en la dirección de un puntero del siguiente modo:

```
int *otro_ptr = *ptr_a_ptr;

printf("%i\n", numero);
printf("%i\n", *ptr_a_numero);
printf("%i\n", *otro_ptr);
printf("%i\n", **ptr_a_ptr);
```

En este caso, los 4 prints imprimen el mismo valor por pantalla (el valor de numero). Los punteros, aunque parezcan engorrosos a primera vista, son muy útiles. Algunos de sus usos más comunes son:

1. **El pasaje de parámetros por referencia**: en C, cuando le pasas un valor a una función, se crea una *copia* de la variable para que la función utilice, mientras que la variable original no se ve alterada. En cambio, si le pasas un puntero, como la función recibe la *dirección de memoria* de la variable, puede cambiar el valor ahí escrito. Esto permite "devolver"varios valores en una función mandándole varios valores por referencia para que los sobreescriba.

2. **La creación de listas de tamaño variable**: en C, normalmente para crear una lista es necesario saber de antemano cuántos objetos vas a guardar en la misma. Pero existen muchas situaciones en las que, o bien no sabemos cuántas cosas vamos a guardar, o la cantidad de cosas a guardar varía dependiendo de las circunstancias. Para eso podemos usar punteros. Más adelante se entrará en detalle sobre las funciones que se usan para hacer esto, pero en resumidas cuentas, se le puede asignar un puntero memoria suficiente para guardar *varios datos de un mismo tipo en la memoria dinámica*, y luego se puede volver a pedir más (o menos) espacio para guardar más (o menos) datos en memoria según sea necesario.

3. **Creación de punteros a función**: en C se pueden crear punteros no sólo que apunten a un tipo de dato, sino que apunten también a una función. Esto es útil si, por ejemplo, tenemos dos bloques de código (p. ej. funciones) que son muy similares menos una parte en el medio. En ese caso podríamos pasar como parámetro una función para que aplique la parte distinta, y así unificar todo en un solo bloque que aplique funciones distintas según sea necesario. Cabe aclarar que para los punteros a función hay que definir que tipos de dato recibe y devuelve la función, por lo que el ejemplo anterior solo funciona si ambas funciones tienen la misma *firma*.

4. **Creación de punteros comodín (void*)**: hasta ahora todos los punteros apuntaban a un tipo de dato *en específico*, pero también se puede tener un puntero a un tipo de dato *cualquiera*, denominado *void**. De este modo también podemos definir funciones que puedan recibir *varios* tipos de dato, en vez de tener que definir una función para cada tipo. Los punteros comodín no pueden ser desreferenciados directamente, antes tienen que ser casteados.

```c
int funcion_1(int var, int foo){
// codigo...
}
int funcion_2(int val, int num){
// codigo distinto...
}
void llenar_lista(int *lista, int *tope, (int) (*funcion)(int, int)){
// mas codigo...
}

int main(){
    // crear listas dinamicas
    int *lista1 = malloc(sizeof(int));
    int tope1 = 0;
    int *lista2 = malloc(sizeof(int));
    int tope2 = 0;

    // crear punteros a funciones
    int (*funcion1)(int, int) = funcion_1;
    int (*funcion2)(int, int) = funcion_2;

    // pasar funciones a otras funciones como [parametro]
    llenar_lista(lista1, &tope1, funcion1);
    llenar_lista(lista2, &tope2, funcion2);

    // anadiendo y accediendo a elementos de listas dinamicas
    lista1 = realloc(lista1, sizeof(int)*tope+2);
    lista1[tope+1] = 22;
    *(lista + 2) = 45;
    tope += 2;

    return 0;
}
```

Listing 1: Ejemplos de punteros (simplificado)

Hasta ahora, en (la mayoría de) los ejemplos dados sólo hemos usado punteros que apuntan al stack . A continuación veremos como podemos crear punteros que apunten al heap con ayuda de ciertas funciones.

2.3. Malloc(), realloc() y free() - accediendo al heap

En la librería estándar de C <**stdlib.h**> existen ciertas funciones que nos permiten asignar y liberar memoria del heap. Nos centraremos en 3, ya que el resto son alguna variación de las mismas:

1. **malloc(tamaño)**, que asigna y devuelve la dirección de un bloque de memoria en el heap de *tamaño* bytes. Con esta función podemos crear un puntero a una variable antes de tener lo que vamos a guardar allí, puesto que solo necesitamos saber cuánto ocupa en memoria (para esto existe la función *sizeof(tipo)*, que devuelve la cantidad de bytes que ocupa un tipo de dato. Cabe destacarse que, en caso de que la función tenga algún error (por ejemplo, no hay memoria suficiente en el dispositivo) devuelve **NULL**, la dirección de memoria *0x00000000*, que es inaccesible bajo circunstancias normales. Por eso, hay que comprobar que cada vez que se utiliza malloc() la función haya devuelto un puntero válido. Como hay que pasarle a malloc() el tamaño en *bytes* que se quiere reservar, también existe la muy útil función *sizeof(tipo_ de_ dato)*, que devuelve precisamente esa información. Si queremos, por ejemplo, reservar memoria para 5 *floats*, podemos hacer la llamada ***malloc(sizeof(float)*5)***.

2. **free(puntero)**, que marca como libre la memoria asociada a *puntero*. Como se mencionó antes, es responsabilidad del programador asegurarse de que **todo** puntero que se asigna con malloc() se libere con free() antes de que termine de ejecutar el programa. Por suerte, existen

varias herramientas de *debugging* que avisan cuándo un programa está asignando memoria pero no la está liberando correctamente.

3. **realloc(ptr_a_reasignar, nuevo_tamaño)**: Reasigna la memoria guardada en el puntero a asignar a un nuevo bloque del tamaño pedido y devuelve el puntero a esa memoria. Para entender bien la utilidad de esta función, hay que pensar en como funcionan las listas dinámicas: reservas con malloc() un espacio de memoria lo suficientemente grande para guardar algunos elementos. Cuando quieras aumentar el tamaño de la lista, vas a tener que: **1.-** asignar nueva memoria con el espacio suficiente, **2.-** copiar todo de una dirección de memoria a la otra, y, **3.-** liberar la memoria que ya no está en uso. La función realloc() hace todo eso de una vez. Como malloc(), realloc() puede llegar a devolver un puntero a **NULL**, por lo que si se asigna un puntero en uso con una llamada de realloc() cabe la posibilidad de que la memoria original quede sin liberar. Por eso, se necesita usar una variable auxiliar para reasignar memoria con realloc().

```c
#include <stdlib.h>

int main(){}
    // crear una lista dinamica de enteros de tamanio 2
    int *vector = malloc(sizeof(int)*2;
    int tope = 0;

    //asignar sus elementos (hay dos formas comunmente usadas)
    vector[tope] = 3;
    tope++;
    *(vector + tope) = 5;
    tope++;

    // asignarle mas espacio usando una variable auxiliar
    int *temp = realloc(vector, sizeof(int)*5);
    // ahora hay que verificar que realloc haya funcionado correctamente
    if (temp != NULL)
        vector = temp;

    // iterando sobre los elementos del vector
    for (int i = 0; i < tope; i++)
        printf("%i\n", vector[i]);

    // destruir el vector antes de terminar
    free(vector);

    return 0;
}
```

Listing 2: Ejemplo de uso correcto de memoria dinámica

3. Detalles de implementación

Para cumplir con lo pedido para el TP, se implementaron dos estructuras **pokemon_t** y **caja_t**. Ambas son *estructuras incompletas*, que implica que la única forma de acceder a sus campos es por medio de funciones implementadas con ese propósito. Por eso, en el programa principal no se crean variables del tipo *pokemon* o *caja*, las funciones reciben y devuelven *pokemon** y *caja**. Esto facilita comprobar que los datos sean válidos, ya que dichas funciones hacen la mayoría de los chequeos y sólo se tiene que comprobar que no devuelvan un valor que represente un error (NULL en el caso de los punteros, 0 en el caso de los ints) A continuación se presenta una breve explicación de las estructuras implementadas:

3.1. Pokemon_t

```c
#define MAX_NOMBRE_POKEMON 30

struct pokemon_t {
    char nombre[MAX_NOMBRE_POKEMON];
    int nivel;
    int ataque;
    int defensa;
};
```

Listing 3: Estructura pokemon_t

Como se puede ver, la estructura de un pokemon en este TP es bastante sencilla. El pokemon tiene un nombre y 3 estadísticas representadas por enteros: nivel, ataque y defensa.

El nombre es un vector estático con un tamaño máximo de 30, que fue acordado previamente. Nivel, ataque y defensa deberían ser no negativos. Como pokemon_t es una estructura incompleta, la única forma de cambiar el valor de estos campos es cuando se crea el pokemon con la función *pokemon_crear_desde_string*, que verifica que esos valores no sean negativos, y los corrige en caso de ser necesario.

En cuanto a funciones relacionadas a los pokemones, también son bastante sencillas. La función *pokemon_crear_desde_string* es la más compleja de todas, ya que tiene que comprobar la validez de varios datos. Primero comprueba que la string recibida no sea nula, luego que la memoria asignada al pokemon a crear tampoco sea nula, luego que se haya leído la cantidad de datos correcta de la string, y por último que los datos numéricos leídos (nivel, ataque y defensa) sean válidos, para finalmente devolver el pokemon.

El resto de las funciones son más simples, ya que sólo devuelven un campo del pokemon, y *pokemon_destruir* destruye al pokemon. Basta con comprobar que el puntero recibido no sea nulo y (en el caso de nivel, ataque y defensa) comprobar que el valor que se va a devolver tenga sentido. Las 5 funciones tienen un aspecto similar a este:

```c
int pokemon_ataque(pokemon_t *pokemon)
{
    int atk = 0;

    if (pokemon != NULL)
        atk = pokemon->ataque;

    if (atk < 0)
        atk = 0;

    return atk;
}
```

Listing 4: Ejemplo

3.2. Caja_t

```
struct caja_t {
    pokemon_t **pokemones;
    int cantidad;
};
```

Listing 5: Estructura caja_t

La estructura de las cajas es aún más sencilla, es simplemente un vector dinámico de punteros a pokemon, ya que no se pedía ninguna otra funcionalidad de la caja más allá de guardar pokemones. La caja guarda punteros a pokemon porque eso es lo que devuelven y reciben las funciones de pokemon.h, que se encarga de manejar pokemones individuales. La cantidad de pokemones también hace las veces de tope del vector dinámico. Como se pedía que cada realloc() aumentara el tamaño de vector de a uno, no hace falta tener una capacidad guardada también.

Cabe aclarar que las cajas siempre están en orden alfabético, ya que las dos únicas funciones que pueden crear y/o agregar elementos a una caja (*caja_cargar_archivo* y *caja_combinar*) las ordenan alfabéticamente como último paso antes de devolver (el puntero a) la caja. El algoritmo de ordenamiento elegido para ordenar alfabéticamente las cajas es el *bubble sort*, simplemente porque es fácil de implementar y, aunque resulte menos eficiente que otros algoritmos de ordenamiento, es suficiente para este caso.

Las funciones de las cajas son un poco más complejas, ya que se está trabajando con vectores dinámicos y hay más valores que comprobar. Para ayudar con la implementación de las funciones pedidas, se crearon 3 funciones privadas auxiliares: una que convierte un pokemon de vuelta a su representación como línea de un csv, una que agrega un pokemon a una caja, y una que ordena una caja en orden alfabético.

La función *caja_cargar_archivo* abre el archivo con el nombre recibido, crea una caja* vacía, y, mientras no haya llegado al final del archivo, lee una línea para pasarla a *pokemon_crear_desde_string* y añade el pokemon a la caja, reasignando la memoria suficiente y actualizando la cantidad de pokemones. Al final de añadir los pokemones, ordena la caja antes de devolverla. Cada vez que se recibe un puntero se chequea que no sea nulo antes de continuar. Si en algún punto una función devuelve algún puntero nulo, la función termina y devuelve la caja hasta donde logró cargar, si cargó algo; o un puntero nulo, si no cargó nada en la caja. También se tuvo precaución de que el archivo siempre se cerrara antes de que la función terminase.

La función *caja_guardar_archivo* funciona de forma similar, aunque es algo más simple por el hecho de no tener que hacer ningún realloc(). Simplemente chequea que los punteros recibidos no sean nulos y luego va copiando en el archivo cada pokemon en una línea y aumentando en 1 un contador cada vez que logra escribir un pokemon para devolver la cantidad escrita al final. *caja_recorrer* funciona de forma similar, solo que en vez de convertir los pokemones a strings para escribir en un archivo, les aplica una función.

Para *caja_combinar* se decidió ir por la implementación sencilla de crear una caja nueva, añadirle todos los pokemones de ambas cajas, y luego ordenarla alfabéticamente. Esto garantiza que el resultado sale en orden alfabético, aunque por alguna razón las cajas a combinar no lo estén.

Las funciones *caja_cantidad*, *caja_obtener_pokemon* y *caja_destruir* son parecidas a las funciones de pokemon.h, simplemente realizan la acción pedida haciendo primero comprobaciones básicas sobre la validez de los datos de entrada y salida.