asmlab

Organización del Computador

Primer Cuatrimestre de 2023

1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre distintas estructuras de datos para que sean utilizadas como base de un juego de cartas tipo solitario. Las estructuras serán de los tipos básicos `Int`, `String` y `Card`, además de dos estructura más complejas como `List` y `Array` que serán utilizadas como contenedores.

Las funciones sobre `Int` permitirán operar sobre estos valores para copiarlos, compararlos e imprimirlos. Las funciones de `String` serán operaciones sobre cadenas de caracteres terminadas en cero. Por último el tipo `Card` modelará una carta de un palo y un número dados.

Los contenedores `List` y `Array` poseerán funciones equivalentes que permitirán modelar un mazo de cartas, tanto en terminos de una lista de cartas, como de un arreglo de cartas. La estructura de `List` permitirá crear, agregar, borrar e imprimir elementos de una lista doblemente enlazada. Mientras que la estructura `Array` implementará un arreglo dinámico con capacidad limitada. Ambas estructuras pueden almacenar valores de cualquier tipo. Por esta razón se agrega a cada estructura un campo que indica el tipo del dato almacenado.

Para poder operar con los datos dentro de los arreglos y listas, se debe conocer qué funciones básicas están asociadas a cada tipo. Para esto se provee un conjunto de funciones que retornan el puntero a la función correspondiente.
Las siguientes son las funciones que permiten obtener la función asociada a cada tipo para una acción dada:

- `funcCmp_t* getCompareFunction(type_t t)`
- `funcClone_t* getCloneFunction(type_t t)`
- `funcDelete_t* getDeleteFunction(type_t t)`
- `funcPrint_t* getPrintFunction(type_t t)`

Cada una de estas funciones toma un tipo de datos y retorna la función asociada a la acción indicada. Los tipos de las funciones de retorno son:

- Función comparar: `typedef int32_t (funcCmp_t)(void*, void*);`
- Función duplicar: `typedef void* (funcClone_t)(void*);`
- Función borrar: `typedef void (funcDelete_t)(void*);`
- Función imprimir: `typedef void (funcPrint_t)(void*, FILE*);`

Los valores posibles de entrada como `type_t` son:

```
typedef enum e_type {
    TypeNone = 0,
    TypeInt = 1,
    TypeString = 2,
    TypeCard = 3
} type_t;
```

2. Estructuras

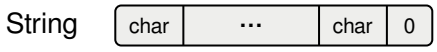
En esta sección se describiran las estructuras de datos a utilizar.

Estructura `Int`

Representa un puntero a un entero con signo de 32 bits.

Estructura `String`

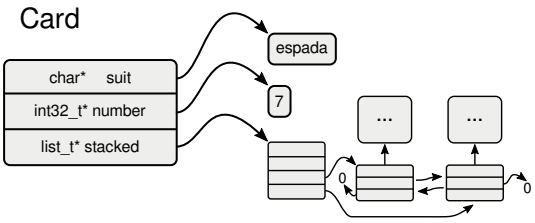
Este tipo no es una estructura en sí misma sino un puntero a un *string* de C terminado con el caracter nulo `'\0'`.



Estructura `Card`

Representa una carta, con un palo indicado como un *string* y un número indicado por un entero. Este tipo carta, además puede modelar todas las cartas que están debajo de esta, almacenando en uno de sus campos una lista de cartas.

La estructura `card_t` contiene un puntero a un *string* dado por `suit`, un puntero a un número dado por `number` y un puntero a una estructura de lista dada por *stacked*. Esta última debe ser una lista de elementos de tipo `card`.

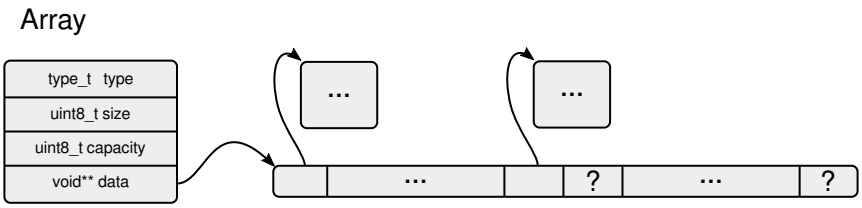



```
typedef struct s_card {
    char*     suit;
    int32_t* number;
    list_t* stacked;
} card_t;
```

Estructura `Array`

Implementa un array dinámico de capacidad limitada. El mismo puede contener como máximo la cantidad de datos indicada en `capacity`. Los datos serán todos del mismo tipo, ya sean *Strings*, enteros o cartas.

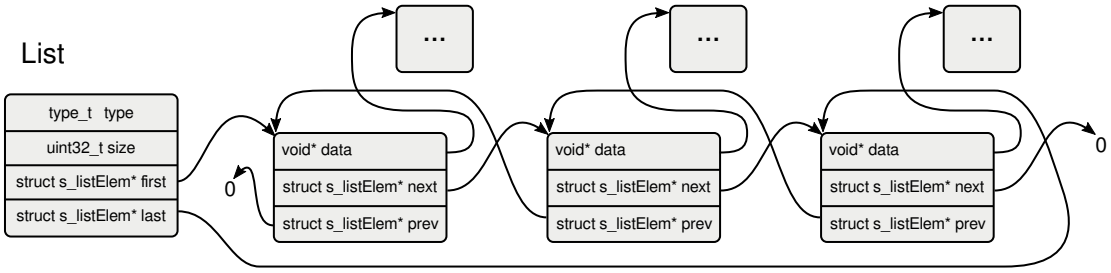
La estructura `array_t` contiene un puntero al arreglo identificado como *data* y la cantidad de elementos ocupados como *size*. Además, se indica el tipo de los datos en `type`.




```
typedef struct s_array {
    type_t  type;
    uint8_t size;
    uint8_t capacity;
    void** data;
} array_t;
```

Estructura `List`

Implementa una lista doblemente enlazada de nodos. La estructura `list_t` contiene un puntero al primer y último elemento de la lista, mientras que cada elemento es un nodo de tipo `listElem_t` que contiene un puntero a su siguiente, anterior y al dato almacenado. Este último es de tipo `void*`, permitiendo referenciar cualquier tipo de datos.

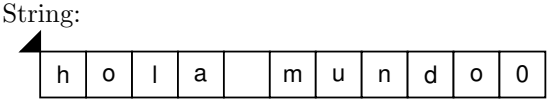



```
typedef struct s_listElem {
    void* data;
    struct s_listElem* next;
    struct s_listElem* prev;
} listElem_t;

typedef struct s_list {
    type_t   type;
    uint32_t size;
    struct s_listElem* first;
    struct s_listElem* last;
} list_t;
```

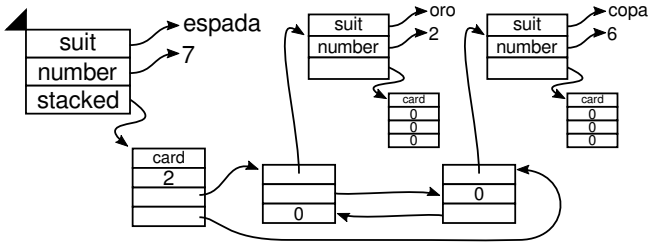
Ejemplos

- String:



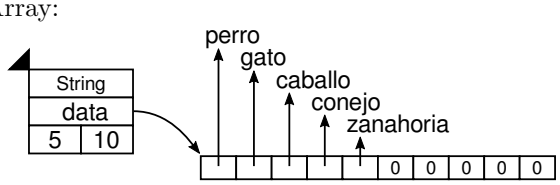
  Print: `hola mundo`

- Card:



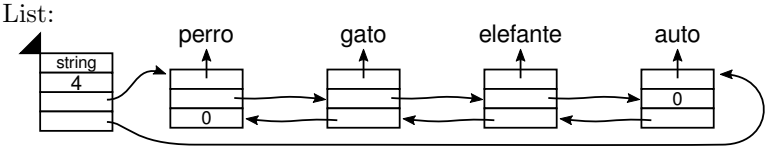
  Print: `{espada-7-[{oro-2-[]},{copa-6-[]}]}`

- Array:



  Print: `[perro,gato,caballo,conejo,zanahoria]`

- List:



Print: `[perro,gato,elefante,auto]`

Juego y estructura `Game`

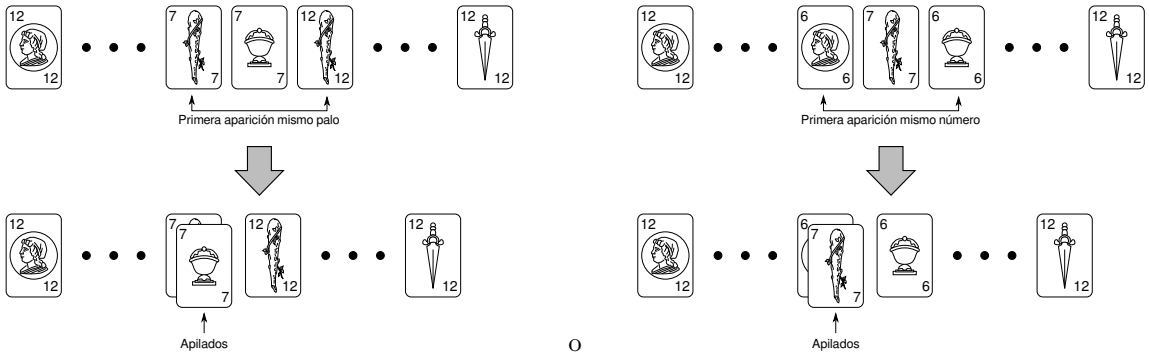
El nucleo de este trabajo práctico, consiste en implementar un juego tipo solitario utilizando todas las funciones sobre estructuras de datos que fueron armadas. El juego comienza desplegando sobre una mesa las cartas una a una, tal que si se cumple una condición estas deban ser apiladas. El objetivo es apilar la mayor cantidad de cartas, si al final se lograron apilar todas las cartas y solo quedan dos pilas, entonces se gana el juego. Caso contrario, si quedan más de dos pilas de cartas sobre la mesa, entonces se perdió el juego.

Los pasos para jugar al juego son los siguientes:

a) Mezclar el mazo de cartas.

b) Colocar inicialmente tres cartas una detrás de la otra en una fila horizontal de derecha a izquierda.

c) Buscar desde el principio de la fila de cartas, una carta tal que tenga a su derecha e izquierda un par de cartas que sean del mismo palo o mismo número. Si la carta cumple la condición, apilar la carta sobre la carta a izquierda del par que cumple la condición.



d) Repetir el procedimiento anterior, siempre buscando desde el principio, hasta que no queden cartas por apilar.

e) Si no hay más cartas que puedan ser apiladas, entonces agregar una nueva carta luego de la última carta agregada.

f) El juego termina cuando no hay más cartas que agregar y no se pueden apilar más cartas.

El juego ya está programado como parte del framework que se provee.[1] La estructura que utiliza el juego para almacenar todos sus datos es la siguiente:

```
typedef struct s_game {
    void*         cardDeck;
    funcGet_t*    funcGet;
    funcRemove_t* funcRemove;
    funcSize_t*   funcSize;
    funcPrint_t*  funcPrint;
    funcDelete_t* funcDelete;
} game_t;
```

[1] Nota que el juego es determinístico, dado un orden del mazo, el resultado del juego siempre será el mismo.

Donde `cardDeck` es el mazo de cartas, y las siguientes son las funciones que se utilizan a la hora de crear un juego para que la aplicación conozca como interactuar con el mazo de cartas. El mazo de cartas por su parte, puede ser implementado tanto como una lista, o como un arreglo.

Las funciones que se proveen son:

- `game_t* gameNew(void* cardDeck, ... )`
  Crea un juego pasando como parámetro el mazo de cartas ya mezclado y un conjunto de funciones para operar.
- `int gamePlayStep(game_t* g)`
  Ejecuta un paso del juego, es decir, busca la carta que cumpla la condición especificada por el juego y la apila.
- `uint8_t gameGetCardDeckSize(game_t* g)`
  Obtiene la cantidad de cartas que están en juego, es decir, las cartas que aun no han sido apiladas.
- `void gameDelete(game_t* g)`
  Borra un juego, borrando todas las estructuras necesarias.
- `void gamePrint(game_t* g, FILE* pFile)`
  Imprime un juego. Está función simplemente llama a imprimir el mazo de cartas usando la función específica al crear el juego.

3. Enunciado

El trabajo práctico consiste en implementar algunas funciones para operar sobre las estructuras de datos dadas. Algunas funciones ya son provistas por la cátedra.
A continuación se detallan las funciones a implementar en lenguaje C.

- `int32_t intCmp(int32_t* a, int32_t* b)`
- `void intDelete(int32_t* a)`
- `void intPrint(int32_t* a, FILE* pFile)`
- `char* strClone(char* a)`
- `void strPrint(char* a, FILE* pFile)`
- `card_t* cardNew(char* suit, int32_t* number)`
- `char* cardGetSuit(card_t* c)`
- `int32_t* cardGetNumber(card_t* c)`
- `list_t* cardGetStacked(card_t* c)`
- `void cardPrint(card_t* c, FILE* pFile)`
- `array_t* arrayNew(type_t t, uint8_t capacity)`
- `void* arrayRemove(array_t* a, uint8_t i)`
- `void arraySwap(array_t* a, uint8_t i, uint8_t j)`
- `void arrayDelete(array_t* a)`
- `void arrayPrint(array_t* a, FILE* pFile)`
- `void listAddLast(list_t* l, void* data)`
- `list_t* listClone(list_t* l)`
- `void* listRemove(list_t* l, uint8_t i)`
- `void listSwap(list_t* l, uint8_t i, uint8_t j)`
- `void listDelete(list_t* l)`
- `void listPrint(list_t* l, FILE* pFile)`

Además las siguientes funciones deben ser implementadas en lenguaje ensamblador.

- `int32_t* intClone(int32_t* a)`
- `uint32_t strLen(char* a)`(7 Inst.)
- `int32_t strCmp(char* a, char* b)`(25 Inst.)
- `void strDelete(char* a)`(1 Inst.)
- `uint8_t arrayGetSize(array_t* a)`(3 Inst.)
- `void arrayAddLast(array_t* a, void* data)`(20 Inst.)
- `void* arrayGet(array_t* a, uint8_t i)`(8 Inst.)
- `list_t* listNew(type_t t)`(10 Inst.)
- `uint8_t listGetSize(list_t* l)`(2 Inst.)
- `void* listGet(list_t* l, uint8_t i)`(13 Inst.)
- `void listAddFirst(list_t* l, void* data)`(27 Inst.)
- `int32_t cardCmp(card_t* a, card_t* b)`(11 Inst.)
- `card_t* cardClone(card_t* c)`(21 Inst.)
- `void cardAddStacked(card_t* c, card_t* card)`(2 Inst.)
- `void cardDelete(card_t* c)`(11 Inst.)

Recordar que cualquier función auxiliar que desee implementar debe estar en lenguaje ensamblador o C según corresponda el lenguaje solicitado. A modo de **referencia**, se indica entre paréntesis la cantidad de instrucciones o líneas de código necesarias para resolver cada una de las funciones.

Compilación y Testeo

Para compilar el código y poder correr las pruebas cortas deberá ejecutar `make main` y luego `./runMain.sh`. En cambio, para compilar el código y correr las pruebas intensivas deberá ejecutar `./runTester.sh`. Adicionalmente el archivo `game.c` contiene la lógica de ejecución del juego. Este se compila de la misma forma `make game` y se puede ejecutar por medio de `./runGame.sh`.

Pruebas cortas

Deberá construirse un programa de prueba completando el archivo `main.c` para que realice las acciones detalladas a continuación, una después de la otra:

1- **Caso array**

- Crear un mazo con 5 cartas sobre un arreglo
- Imprimir el mazo.
- Apilar una carta cualquiera del mazo sobre otra carta cualquiera.
- Imprimir nuevamente el mazo.
- Borrar el mazo.

2- **Caso list**

- Realizar las mismas acciones que para el arreglo.

El programa puede correrse con `./runMain.sh` para verificar que no se pierde memoria ni se realizan accesos incorrectos a la misma.

Pruebas intensivas (Testing)

Entregamos también una serie de *tests* o pruebas intensivas para que pueda verificarse el buen funcionamiento del código de manera automática. Para correr el testing se debe ejecutar `./runTester.sh`, que compilará el *tester* y correrá todos los tests de la cátedra. Un test consiste en la creación, inserción, eliminación, ejecución de funciones e impresión en archivos de alguna estructura implementada. Luego de cada test, el *script* comparará los archivos generados por su TP con las soluciones correctas provistas por la cátedra. También será probada la correcta administración de la memoria dinámica.

Notas

- Toda la memoria dinámica reservada usando la función `malloc` debe ser correctamente liberada, utilizando la función `free`.
- Para el manejo de archivos se recomienda usar las funciones de `C`: `fopen`, `fprintf`, `fwrite`, `fputc`, `fputs` y `fclose`.
- Para el manejo de strings **no está permitido** usar las funciones de `C`: `strlen`, `strcpy`, `strcmp` y `strdup`.
- Para correr los tests deben tener instalado *Valgrind* (en Ubuntu: `sudo apt-get install valgrind`).
- Para corregir los TPs usaremos los mismos tests que les fueron entregados. Es condición necesaria para la aprobación que el TP supere correctamente todos los tests.

Archivos

Se entregan los siguientes archivos:

- `lib.h`: Contiene la definición de las estructuras y de las funciones a realizar. No debe ser modificado.
- `lib.s`: Archivo a completar con la implementación de las funciones en lenguaje ensamblador.
- `lib.c`: Archivo a completar con la implementación de las funciones en lenguaje C.
- `Makefile`: Contiene las instrucciones para ensamblar y compilar el código.
- `main.c`: Archivo donde escribir los ejercicios para las pruebas cortas (`runMain.sh`).
- `tester.c`: Archivo del tester de la cátedra. No debe ser modificado.
- `game.c`: Archivo del juego. No debe ser modificado.
- `runMain.sh`: Script para ejecutar el test simple (pruebas cortas). No debe ser modificado.
- `runTester.sh`: Script para ejecutar todos los tests intensivos. No debe ser modificado.
- `runGame.sh`: Script para ejecutar el juego. No debe ser modificado.
- `salida.catedra.caso*.txt`: Archivos de salida para comparar con sus salidas. No debe ser modificado.

4. Forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar el mismo contenido que fue dado para realizarlo, habiendo modificado **solamente** los archivos `lib.s`, `lib.c` y `main.c`.

Deberá ser entregado a través del repositorio GIT del grupo, haciendo un PR desde su branch de trabajo (con el nombre que ustedes elijan) hacia el branch **asmlab**. Por favor, no hacer PR hacia el branch **main**.

Una vez hecho esto, se ejecutarán los tests intensivos en un servidor y si son satisfactorios, será indicado por la plataforma. Agreguen como revisores al equipos docente en caso de que el test sea satisfactorio y con eso queda la entrega realizada.

5. Anexo

Esta sección describe la operatoria de todas las funciones, ya sean implementadas por la cátedra o solicitadas como parte del trabajo práctico.

Estructura `Int`

- `int32_t intCmp(int32_t* a, int32_t* b)`
  Compara los valores de `a` y `b`, y retorna:
  - 0 si son iguales
  - 1 si `a` $<$ `b`
  - -1 si `b` $<$ `a`
- `int32_t* intClone(int32_t* a)`
  Solicita 4 bytes de memoria donde copiar el dato almacenado en `a`.
- `void intDelete(int32_t* a)`
  Libera la memoria que contiene el dato pasado por parámetro. Equivalente a la función *free*.
- `void intPrint(int32_t* a, FILE* pFile)`
  Imprime el valor entero en el *stream* indicado a través de `pFile`.

Estructura `String`

- `char* strClone(char* a)`
  Genera una copia del *string* pasado por parámetro. El puntero pasado siempre es válido aunque podría corresponderse a la *string* vacía.
- `uint32_t strLen(char* a)`
  Retorna la cantidad de caracteres distintos de cero del *string* pasado por parámetro.
- `int32_t strCmp(char* a, char* b)`
  Compara dos *strings* en orden lexicográfico[2]. Debe retornar:
  - 0 si son iguales
  - 1 si `a` $<$ `b`
  - -1 si `b` $<$ `a`
- `void strDelete(char* a)`
  Borra el *string* pasado por parámetro. Esta función es equivalente a la función `free`.
- `void strPrint(char* a, FILE* pFile)`
  Escribe el *string* en el *stream* indicado a través de `pFile`. Si el *string* es vacío debe escribir "`NULL`".

Estructura `Card`

- `card_t* cardNew(char* suit, int32_t* number)`
  Crea una nueva `card_t` con el `suit` y el `number` pasados por parámetro. Debe hacer una copia de los argumentos.
- `char* cardGetSuit(card_t* c)`
  Devuelve un puntero al valor del suit de la carta `c`
- `int32_t* cardGetNumber(card_t* c)`
  Devuelve un puntero al valor del número de la carta `c`
- `list_t* cardGetStacked(card_t* c)`
  Devuelve un puntero a la lista de cartas apiladas bajo la carta `c`
- `int32_t cardCmp(card_t* a, card_t* b)`
  Realiza una comparación entre cartas. En primer lugar, compara en orden lexicográfico los suits y devuelve:

[2] `https://es.wikipedia.org/wiki/Orden_lexicografico`

- 1 si `a->suit` < `b->suit`
- -1 si `b->suit` < `a->suit`

Si son iguales los suits, se procede a comparar número y devuelve:

- 0 si son iguales
- 1 si `a->number` < `b->number`
- -1 si `b->number` < `a->number`

- `card_t* cardClone(card_t* c)`
  Genera una copia de la carta `c`, incluyendo copias de `suit`, de `number` y de las cartas apiladas debajo de `c`.
- `void cardAddStacked(card_t* c, card_t* card)`
  Agrega una copia de la carta `card` a las cartas apiladas debajo de `c`. La carta debe ser agregada al comienzo de dicha pila.
- `void cardDelete(card_t* c)`
  Borra la carta `c`. Debe llamarse a la función delete de todos los elementos de la carta.
- `void cardPrint(card_t* c, FILE* pFile)`
  Escribe la carta *c* en el *stream* indicado a través de `pFile`. El formato de impresión tiene la forma: {*suit-number-list_of_stacked_cards*}

Estructura `Array`

- `array_t* arrayNew(type_t t, uint8_t capacity)`
  Crea un array nuevo con elementos de tipo `t` y capacidad indicada por `capacity`.
- `uint8_t arrayGetSize(array_t* a)`
  Obtiene la cantidad de elementos ocupados del arreglo.
- `void arrayAddLast(array_t* a, void* data)`
  Agrega un elemento al final del arreglo. Si el arreglo no tiene capacidad suficiente, entonces no hace nada. Esta función debe hacer una copia del dato.
- `void* arrayGet(array_t* a, uint8_t i)`
  Obtiene el i-esimo elemento del arreglo, si `i` se encuentra fuera de rango, retorna 0.
- `void* arrayRemove(array_t* a, uint8_t i)`
  Quita el i-esimo elemento del arreglo, si `i` se encuentra fuera de rango, retorna 0. El arreglo es reacomodado de forma que ese elemento indicado sea quitado y retornado.
- `void arraySwap(array_t* a, uint8_t i, uint8_t j)`
  Invierte el contenido del i-ésimo elemento con el j-ésimo elemento. Si alguno de los dos índices se encuentra fuera de rango, no realiza ninguna acción.
- `void arrayDelete(array_t* a)`
  Borra el arreglo, para esto borra todos los datos que el arreglo contenga, junto con las estructuras propias del tipo arreglo.
- `void arrayPrint(array_t* a, FILE* pFile)`
  Escribe en el *stream* indicado por `pFile` el arreglo almacenada en `a`. Para cada dato llama a la función de impresión del dato correspondiente. El formato del arreglo será: `[`x_0`,...,`x_{n-1}`]`, suponiendo que x_i es el resultado de escribir el i-ésimo elemento.

Estructura `List`

- `list_t* listNew(type_t t)`
  Crea una nueva `list_t` vacía donde los punteros a `first` y `last` estén inicializados en cero. Además toma el tipo de datos que almacenará la lista, este dato será utilizado por las diferentes funciones para obtener las funciones que permiten operar con los datos almacenados.
- `uint8_t listGetSize(list_t* l)`
  Obtiene la cantidad de elementos de la lista.

- `void listAddFirst(list_t* l, void* data)`
  Agrega un nuevo nodo al principio de la lista, que almacene `data`. Esta función debe hacer una copia del dato.
- `void listAddLast(list_t* l, void* data)`
  Agrega un nuevo nodo al final de la lista, que almacene `data`. Esta función debe hacer una copia del dato.
- `void* listGet(list_t* l, uint8_t i)`
  Obtiene el i-esimo elemento de la lista, si `i` se encuentra fuera de rango, retorna 0.
- `void* listRemove(list_t* l, uint8_t i)`
  Quita el i-esimo elemento de la lista, si `i` se encuentra fuera de rango, retorna 0. El nodo en la lista es borrado y el elemento retornado por la función.
- `void listSwap(list_t* l, uint8_t i, uint8_t j)`
  Invierte el contenido del i-esimo elemento con el j-esimo elemento de la lista. Si alguno de los dos indices se encuentra fuera de rango, no realiza ninguna acción.
- `list_t* listClone(list_t* l)`

- `void listDelete(list_t* l)`
  Borra la lista completa con todos sus nodos. Para borrar los datos utiliza la función delete asociada al tipo de datos almacenado en la lista.
- `void listPrint(list_t* l, FILE* pFile)`
  Escribe en el *stream* indicado por `pFile` la lista almacenada en `l`. Para cada dato llama a la función de impresión del dato correspondiente. El formato de la lista será: `[`x_0`,...,`x_{n-1}`]`, suponiendo que x_i es el resultado de escribir el i-ésimo elemento.