Algoritmos y Estructuras de Datos II

TALLER - 5 de mayo 2022

Laboratorio 4: Tipos Abstractos de Datos (TADs)

- Revisión 2020: Leandro Ramos
- Revisión 2021: Marco Rocchietti
- Revisión 2022: Marco Rocchietti

Objetivos

1. Profundizar uso de punteros y memoria dinámica
2. Llevar a lenguaje C los conceptos de TAD estudiados en el Teórico-Práctico
3. Comprender conceptos de encapsulamiento vs acoplamiento
4. Comprender concepto de implementación opaca
5. Administración de memoria dinámica (`malloc()`, `calloc()`, `free()`)

Ejercicio 0: Punteros++, Arreglos y Cadenas

Operaciones sobre punteros

Se vio que un puntero es un tipo de variable especial que guarda una dirección de memoria. También se mostraron dos operaciones básicas relacionadas con punteros:

- Desreferenciación (**`*`**): obtiene el **valor** de lo *apuntado* por el puntero. Si se tiene una variable de tipo **`int *`** llamada **`p`**, entonces la expresión **`*p`** retornará el valor entero que se aloja en dirección de memoria **`p`**. Se puede usar **`*p`** en el lado izquierdo de una asignación para cambiar el valor que está apuntado por **`p`** (la dirección de memoria guardada en **`p`** se mantiene idéntica)
- Referenciación (**`&`**): obtiene la dirección de memoria de una variable. Si se tiene una variable entera **`x`** declarada como **`int x;`** entonces la expresión **`&x`** es de tipo puntero a **`int`** (es decir que es de tipo **`int *`**) y apunta a la dirección de memoria de la variable **`x`**.

En C además para las variables de tipo puntero se puede usar las operaciones de indexación y el operador flecha (`->`):

- Indexación (p**`[n]`**): Permite obtener el valor que hay en la memoria moviéndose **`n`** lugares hacia adelante desde la dirección de memoria guardada en p. Entonces por ejemplo **`p[0]`** es equivalente a **`*p`**. Cuando se indexa un puntero se debe tener total seguridad de que se va a acceder a memoria asignada a nuestro programa, de lo contrario ocurrirá un *segmentation fault* (viloación de segmento).
- Acceso indirecto (**`->`**): Si **`p`** es un puntero a una estructura **`p->member`** es un atajo a **`(*p).member`** (asumiendo que la estructura tiene un campo llamado **`member`**).

Los valores de las variables del tipo puntero (las direcciones de memoria) se pueden visualizar. Por lo general esto no se hace, salvo a veces para hacer *debug*. La manera es usando **`printf()`** con **`%p`**:

```
int *p=NULL;
int a=55;
p = &a;
printf("La dirección de memoria apuntada por p es: %p", p);
```

El resultado va a ser un número en hexadecimal (con prefijo `0x`…), por ejemplo:

```
La dirección de memoria apuntada por p es: 0x7ffcd9183bdd
```

Cuando se declara una variable de tipo arreglo,

```
int arr[10];
```

hay dos formas de obtener la dirección de memoria al primer elemento:

- Usando el operador de referenciación: **`&arr[0]`**
- Usando el nombre del arreglo: **`arr`**

```
int arr[10];
int *p=NULL;
p = &arr[0]; // Usando operador &
p = arr;     // Usando directamente el nombre de variable del arreglo
```

Alocación de memoria

En el lenguaje del teórico práctico se usa el procedimiento **`alloc()`** para reservar memoria para un puntero, y **`free()`** para liberar dicha memoria:

```
var p: pointer to int

alloc(p)
*p := 5
free(p)
```

En C esto se hace usando las funciones **`malloc()`** y **`free()`**:

```
int *p=NULL;
p = malloc(sizeof(int));
*p = 5;
free(p);
```

La función **malloc()** toma un parámetro de tipo **size_t** (muy parecido a **unsigned int**) que es la cantidad de memoria en *bytes* que se solicita reservar. A diferencia del **alloc()** del teórico, que automáticamente reserva la cantidad necesaria según el tipo de puntero, en C hay que indicar explícitamente la cantidad de *bytes* a reservar. El operador **sizeof()** devuelve la cantidad de *bytes* ocupados por una expresión o tipo, por lo que resulta indispensable para el uso de **malloc()**.

```
$ man malloc
```

El ejercicio consiste en:

a) Modificar el programa en **array.c** para que mediante el puntero **p** se inicialice en cero el arreglo **arr** sin utilizar los operadores **&** y *.

b) Programar la función

```
void set_name(name_t new_name, data_t *d);
```

que debe cambiar el campo **name** de la estructura apuntada por **d** con el contenido de **new_name** y utilizarla para modificar la variable **messi** de tal manera que en su campo **name** contenga la cadena **"Lionel Messi".**

c) Completar el archivo **sizes.c** para que muestre el tamaño en *bytes* de cada miembro de la estructura **data_t** por separado y el tamaño total que ocupa la estructura en memoria. ¿La suma de los miembros coincide con el total? ¿El tamaño del campo **name** depende del nombre que contiene?

d) En el directorio **static** se encuentra el programa del Laboratorio 1 que carga en un arreglo en memoria estática desde un archivo. Completar en la carpeta **dynamic** la función **array_from_file()** de **array_helpers.c**:

```
int *array_from_file(const char *filepath, size_t *length);
```

que carga los datos del archivo **filepath** devolviendo un puntero a memoria dinámica con los elementos arreglo y dejando en ***length** la cantidad de elementos leídos. Completar además en **main.c** el código necesario para liberar la memoria utilizada por el arreglo. Probar el programa con todos los archivos de la carpeta **input** para asegurar el correcto funcionamiento.

Preliminares - TADS

Encapsulamiento

Lo primero que debemos observar es la forma en la que logramos mantener separadas la especificación del TAD de su implementación. Cuando definimos un TAD es deseable garantizar **encapsulamiento**, es decir, que solamente se pueda acceder y/o modificar su estado a través de las operaciones provistas. Esto no siempre es trivial ya que los tipos abstractos están implementados en base a los tipos concretos del lenguaje. Entonces es importante que además de separar la especificación e implementación se

garantice que quién utilice el TAD no pueda acceder a la representación interna y operar con los tipos concretos de manera descontrolada.

No todos los lenguajes brindan las mismas herramientas para lograr una implementación *opaca* y se debe usar el mecanismo apropiado según sea el caso. Particularmente el lenguaje del teórico-práctico separa la especificación de un TAD de su implementación utilizando las signaturas **`spec` … `where`** e **`implement` … `where`** respectivamente. En este laboratorio se debe buscar la manera de lograr encapsulamiento usando el lenguaje **C**.

Métodos de TADs

En el diseño de los tipos abstractos de datos (tal como se vio en el teórico-práctico) aparecen los **constructores**, las **operaciones** y los **destructores,** que se declaran como funciones o procedimientos. Recordar (se vio en el laboratorio anterior) que los procedimiento en C no existen como tales sino que se usan funciones con tipo de retorno `void`, es decir, funciones que no devuelven ningún valor al llamarlas. A veces se buscará evitar procedimientos con una variable de salida usando directamente una función para simplificar y evitar así usar punteros extra (en el ejercicio 4 del laboratorio 3 se vio que es necesario usar punteros para simular variables de salida).

A diferencia del práctico, a las *precondiciones* y *postcondiciones* de los métodos **sí vamos a verificarlas** (en la medida de lo posible). Recuerden que nuestros programas deben ser **robustos**, por lo tanto cuando corresponda usaremos `assert()` para garantizar el cumplimiento de las pre y post condiciones de los métodos. Esta práctica es propia de la etapa de desarrollo de un programa, y una vez que el mismo está finalizado, verificado y listo para desplegarlo en producción, se pueden eliminar las aserciones mediante un flag de compilación.

Ejercicio 1: TAD Par

Considerar la siguiente especificación del TAD Par

---

spec Pair **where**

constructors
  fun new(**in** x : int, **in** y : int) **ret** p : Pair
  {- crea un par con componentes (x, y) -}

destroy
  proc destroy(**in/out** p : Pair)
  {- libera memoria en caso que sea necesario -}

operations
  fun first(**in** p : Pair) **ret** x : int
  {- devuelve el primer componente del par-}

  fun second(**in** p : Pair) **ret** y : int
  {- devuelve el segundo componente del par-}

  fun swapped(**in** p : Pair) **ret** s : Pair
  {- devuelve un nuevo par con los componentes de p intercambiados -}

---

a) Abrir la carpeta **`pair_a`** y revisar la especificación del TAD en **`pair.h`**. Luego completar la implementación de las funciones y compilar usando el módulo **`main.c`** como programa de prueba. *¿La implementación logra encapsulamiento? ¿Por qué sí? ¿Por qué no?*

b) Abrir la carpeta **`pair_b`** y revisar la especificación del TAD en **`pair.h`**. Luego completar la implementación de las funciones y compilar usando el módulo **`main.c`** como programa de prueba. *¿La implementación logra encapsulamiento? ¿Por qué sí? ¿Por qué no?*

IMPORTANTE: Para definir constructores, destructores y operaciones de copia será necesario hacer manejo de memoria dinámica (pedir y liberar memoria en tiempo de ejecución). En este caso se necesita espacio suficiente para almacenar un valor de tipo `struct _pair_t`.

c) Abrir la carpeta **`pair_c`** y revisar **`pair.h`**. Copiar el archivo **`pair.c`** del apartado **(b)** y agregar las definiciones necesarias para que funcione con la nueva versión de **`pair.h`**. *¿La implementación logra encapsulamiento?* Copiar el archivo **`main.c`** del apartado anterior y compilar. Hacer las modificaciones necesarias en **`main.c`** para que compile sin errores.

d) Considerar la nueva especificación polimórfica para el TAD Pair:

---

spec Pair **of** T **where**

constructors
  fun new(**in** x : T, **in** y : T) **ret** p : Pair **of** T
  {- crea un par con componentes (x, y) -}

destroy
  proc destroy(**in/out** p : Pair **of** T)
  {- libera memoria en caso que sea necesario -}

operations
  fun first(**in** p : Pair **of** T) **ret** x : T
  {- devuelve el primer componente del par-}

  fun second(**in** p : Pair **of** T) **ret** y : T
  {- devuelve el segundo componente del par-}

  fun swapped(**in** p : Pair **of** T) **ret** s : Pair **of** T
  {- devuelve un nuevo par con los componentes de p intercambiados -}

¿Qué diferencia hay entre la especificación anterior y la que se encuentra en el **`pair.h`** de la carpeta **`pair_d`**? Copiar **`pair.c`** del apartado anterior y modificarlo para utilizar la nueva interfaz especificada en **`pair.h`**. Pueden utilizar el **`main.c`** del apartado anterior para compilar.

Ejercicio 2: TAD Contador

Dentro de la carpeta **`ej2`** se encuentran los siguientes archivos:

Archivo	*Descripción*
`counter.h`	Contiene la especificación del TAD Contador.
`counter.c`	Contiene la implementación del TAD Contador.
`main.c`	Contiene al programa principal que lee uno a uno los caracteres de un archivo chequeando si los paréntesis están balanceados.

a) Implementar el TAD Contador. Para ello deben abrir **`counter.c`** y programar cada uno de los constructores y operaciones cumpliendo la especificación dada en **`counter.h`**. Recordar que deben verificar en **`counter.c`** todas las precondiciones especificadas en **`counter.h`** usando llamadas a la función `assert()`.

b) Usar el TAD Contador para chequear paréntesis balanceados. Para ello deben abrir el archivo **`main.c`** y entender qué es lo que hace la función `matching_parentheses()` y completar con llamadas al constructor y destructor del contador donde consideren necesario. ¡Es muy importante llamar al destructor del TAD una vez este no sea necesario para poder liberar el espacio de memoria que tiene asignado!

Una vez implementados los incisos **(a)**, **(b)** compilar ejecutando:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c counter.c main.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 counter.o main.o -o counter
```

Ahora se puede ejecutar el programa corriendo:

```
$ ./counter input/<file>.in
```

siendo **`<file>`** alguno de los nombres de archivo dentro de la carpeta **`input`**. Asegurarse que para aquellos archivos con paréntesis balanceados, al ejecutar el programa se imprima en pantalla

```
Parentheses match.
```

y para aquellos con paréntesis no balanceados imprima

```
Parentheses mismatch.
```

Ejercicio 3: TAD Lista

Dentro de la carpeta **`ej3`** se encuentran los siguientes archivos:

Archivo	*Descripción*
`main.c`	Contiene al programa principal que lee los números de un archivo para ser cargados en nuestra lista y obtener el promedio.
`array_helpers.h`	Contiene descripciones de funciones auxiliares para manipular arreglos.
`array_helpers.c`	Contiene implementaciones de dichas funciones.

a) Crear un archivo **`list.h`**, especificando allí todos los constructores y operaciones vistos sobre el TAD Lista en el teórico. Recomendamos definir el nombre del TAD como `list` ya que en el archivo **`main.c`** se encuentra mencionado de esa manera.

Existe un par de diferencias entre nuestro TAD Lista en C respecto al visto en el teórico. Para simplificar la implementación, nuestras listas serán solamente de tipo `int`, es decir, no hay *polimorfismo*. Si bien el tipo será fijo (`int`), una buena idea es definir un tipo en **`list.h`** usando `typedef`. Un ejemplo de esto sería definir

```
typedef int list_elem;
```

y utilizar `list_elem` en vez de `int` en todos los constructores/operaciones (al estilo de lo realizado en el ejercicio ***1d)***).

Otra diferencia con el teórico es que aquellos procedimientos que modifiquen la lista deben escribirse como funciones que devuelvan la lista resultante. Como ya fue mencionado, esto es para evitar tener que simular parámetros de salida.

No olvidar de:

- Garantizar encapsulamiento en tu TAD.
- Especificar una función de destrucción y copia.
- Especificar las precondiciones.

b) Crear un archivo **`list.c`**, e implementar cada uno de los constructores y operaciones declaradas en el archivo **`list.h`** . La implementación debe ser como se presenta en el teórico, es decir, utilizando punteros (listas enlazadas).

c) Abrir el archivo **`main.c`** e implementar las funciones `array_to_list()` y `average()`. Para la implementación de `average()` te sugerimos que revises la definición del teórico.

Una vez implementados los incisos **a)**, **b)** y **c)**, compilar ejecutando:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c list.c array_helpers.c main.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 list.o array_helpers.o main.o -o average
```

Ahora se puede ejecutar el programa corriendo:

```
$ ./average input/<file>.in
```

siendo **<file>** alguno de los nombres de archivo dentro de la carpeta **input**. Asegurar que el valor de los promedios que se imprimen en pantalla sean correctos y animense a definir sus propios casos de *input*.