Algoritmos y Estructuras de Datos II

TALLER - 4 de mayo 2021

Laboratorio 4: Tipos Abstractos de Datos (TADs)

- Revisión 2021: Marco Rocchietti
- Revisión 2020: Leandro Ramos
- Revisión 2018: Gonzalo Peralta

Objetivos

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

Preliminares

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 deben buscar la manera de lograr encapsulamiento usando el lenguaje **C**.

Métodos de TADs

En el diseño de los tipos abstractos de datos (tal como vieron en el teórico-práctico) aparecen los **constructores**, las **operaciones** y los **destructores,** que se declaran como funciones o procedimientos. Recuerden (se vio en el laboratorio anterior) que los procedimiento en C no existen como tales sino que usamos funciones con tipo de retorno `void`, es decir, funciones que no devuelven ningún valor al llamarlas. A veces buscaremos 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 vimos 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.

Ejercicio 1: TAD Par

Consideren 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). Recomendamos que investiguen el uso de `malloc()` y `free()`. Pueden consultar las páginas de manual ejecutando

```
$ man malloc
```

para entender cómo deberían usar esa función. A diferencia del `alloc()` del teórico, la función `malloc()` recibe como argumento la cantidad de memoria (en *bytes*) que se necesita. En nuestro caso necesitaremos espacio suficiente para almacenar un valor de tipo `struct _pair_t`. Para obtener el tamaño de un tipo en C recomendamos utilizar la función `sizeof()`.

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

Ahora se puede ejecutar el programa corriendo:

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

wiendo **<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** vas a encontrar 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 soportaremos *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 te olvides 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
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o average *.o main.c
```

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*.