Algoritmos y Estructuras de Datos II

Laboratorio 5: Stack

Objetivos

1. Familiarizarse con cadenas en C
2. Creación de arreglos dinámicos
3. Contrastar *memoria dinámica* vs *memoria estática* (*heap* vs *stack*)
4. Implementa el TAD Pila con distintas representaciones
5. Concepto y verificación de invariantes de representación
6. Funciones `calloc()` y `realloc()`
7. Usar **`valgrind`** para detectar problemas de manejo de memoria
8. Usar **`gdb`** para ayudar a solucionar *bugs* de programación

Ejercicio 0: Cadenas y memoria

Las cadenas en C se implementan como arreglos de caracteres. Los caracteres son valores del tipo `char` (que representa exactamente un caracter de *1 byte*), entonces para guardar un *string* en C se puede usar el siguiente arreglo:

```
char cadena[5];
```

En este ejemplo, `cadena` tiene capacidad para guardar un *string* de hasta 4 (cuatro) caracteres de longitud. Esto es así porque toda cadena en C **debe terminar con el caracter** `'\0'`, por lo cual ya tenemos un lugar ocupado. Entonces se puede almacenar en `cadena` una palabra con longitud de entre uno y cuatro caracteres, pero incluso también se puede guardar una palabra vacía (en ese caso **`cadena[0] = '\0'`**). Si queremos armar el *string* con la palabra *"hola"* podemos hacer:

```
char cadena[5]={'h', 'o', 'l', 'a', '\0'};
printf("cadena: %s\n", cadena);
```

Otra forma más cómoda es hacer algo como:

```
char cadena[5]="hola";
printf("cadena: %s\n", cadena);
```

en este caso el caracter `'\0'` se agrega implícitamente en el arreglo `cadena`. Para no tener que contar la cantidad de caracteres que necesitamos se puede definir una cadena directamente haciendo:

```
char cadena[]="hola mundo!";
printf("cadena: %s\n", cadena);
```

el contenido del *array* es el siguiente:

cadena:	'h'	'o'	'l'	'a'	' '	'm'	'u'	'n'	'd'	'o'	'!'	'\0'
	0	1	2	3	4	5	6	7	8	9	10	11

Como ya se vio, los arreglos y los punteros son cosas muy parecidas en C, por lo que también es válido hacer los siguiente:

```
char *cadena="hola mundo!";
printf("cadena: %s\n", cadena);
```

Las siguientes consignas deben resolverse sin utilizar funciones de las librerías estándar **<string.h>** ni **<strings.h>** (salvo en el apartado b):

a) Crear una librería **strfuncs** que incluya las siguientes funciones:

```
size_t string_length(const char *str);
```

que calcula la longitud de la cadena apuntada por **str**, y la función

```
char *string_filter(const char *str, char c);
```

que devuelve una nueva cadena en memoria dinámica que se obtiene tomando los caracteres de **str** que son distintos del caracter **c**.

Luego compilar junto con **main.c** de tal manera que la salida del programa sea:

```
$ ./main
original: 'h.o.l.a m.u.n.d.o.!' (19)
filtrada: 'hola mundo!' (11)
```

b) Compilar y ejecutar el programa implementado en **readstr.c**. Identificar los problemas generados por **scanf()** y reemplazar su uso por la función **fgets()** (consultar las páginas de manual: **man fgets**). Modificar la cadena escrita por **fgets()** para eliminar el **'\n'** agregado al final. Para ello se puede utilizar la función **strlen()** de **string.h** y cualquier otra que resulte de utilidad.

c) Analizar la función **string_clone()** y determinar cuál es el problema en su implementación (si a simple vista no aparece el error, utilizar `valgrind`). Luego modificarla

para que funcione correctamente. Modificar la función **`main()`** para que no queden *memory leaks*. Una vez completada la función **`string_clone()`** verificar qué sucede si se cambia el tipo de la variable **`original`** por:

```
char *original="\n"
               (:)
```

d) Completar la función

```
char *string_clone(const char *str);
```

que debe devolver una copia de la cadena apuntada por **`str`** en nueva memoria dinámica. Para hacerlo usar funciones de la librería **`<string.h>`** a excepción de **`strdup()`**.

Preliminares: Invariantes de representación

Según la representación elegida, los valores de la estructura interna de un TAD deben cumplir ciertas propiedades. Pensemos por ejemplo en el TAD Pair (las versiones del ***1c*** o ***1d*** con punteros del *lab04*). Desde que se crea hasta que se destruye una instancia `p`

```
p = pair_new(x, y);
(:)
p = pair_destroy(p);
```

debe cumplirse que **`p != NULL`**, ya que de otra manera alguna de las operaciones del TAD van a fallar. Por ejemplo **`pair_first()`**, que puede estar implementado como:

```
pair_t pair_first(pair_t p) {
    return p->fst;
}
```

en caso de que **`p==NULL`** la ejecución de esta función generará un *segmentation fault* (violación de segmento) pues no se puede desreferenciar a **`NULL`**.

Otro ejemplo es en la implementación de listas que se pide en el *ejercicio 2* del *Práctico 2.2*:

```
implement List of T where

type List of T = tuple
                   elems: array[1..N] of T
                   size: nat
                 end tuple
```

En este caso una instancia `ls` del TAD List debe cumplir necesariamente **`ls.size <= N`** ya que de otra manera significa que alguna operación generó un error en la estructura interna de representación.

Este tipo de propiedades, entonces, deben **mantenerse invariantes durante toda la vida de la instancia** de un TAD y es una buena idea verificarlas al principio y al final de cada función que implemente una operación. A partir de ahora llamaremos *invariante de*

representación a las propiedades que debe cumplir una instancia para ser válida según el diseño de la implementación. Debe quedar claro que, así como se oculta la implementación al usuario de nuestro TAD, también la invariante de representación será privada.

En este proyecto se va a verificar la *invariante de representación* usando `assert()`. En algunas ocasiones, la representación puede tener una invariante trivial, como por ejemplo la implementación del TAD Lista del laboratorio 4, para la cual no hay ningún chequeo que hacer (¿o sí?). Entonces en los casos que corresponda se debe definir una función *booleana* `invrep()` que verifique que la instancia que se le pasa como parámetro cumpla las propiedades de la invariante. En cada operación del TAD se debe asegurar con `assert()` que la invariante sea verdadera al inicio y al finalizar la operación ¿Y en los constructores y destructores qué pasa con la invariante?

Ejercicio 1: Stack con nodos

a) Implementar el TAD Stack usando listas enlazadas de nodos. Sus operaciones:

Función	Descripción
`stack stack_empty()`	Crea una pila vacía
`stack stack_push(stack s, stack_elem e)`	Inserta un elemento al tope de la pila
`stack stack_pop(stack s)`	Remueve el tope de la pila
`unsigned int stack_size(stack s)`	Obtiene el tamaño de la pila
`stack_elem stack_top(stack s)`	Obtiene el tope de la pila, sin remover. Sólo aplica a una pila no vacía; usar la función `assert()` para verificar esa precondición.
`bool stack_is_empty(stack s)`	Verifica si la pila está vacía.
`stack_elem *stack_to_array(stack s)`	Crea un arreglo con todos los elementos de la pila. El tope de la pila debe quedar en el último elemento del arreglo. Es decir, leyendo el arreglo de derecha a izquierda se obtiene la pila original. Si la pila está vacía, devuelve `NULL`. Para crear el arreglo nuevo usar `calloc()`.
`stack stack_destroy(stack s)`	Libera todos los nodos de la pila.

Como primer paso se debe definir la estructura `struct _s_stack` en el archivo **`stack.c`** al estilo de los nodos del TAD Lista (un campo para el elemento y un puntero al siguiente nodo). Los elementos de la pila serán números enteros (ver la definición del tipo `stack_elem`). Luego continuar con la implementación de las funciones declaradas en **`stack.h`**.

Se debe además crear un archivo de prueba **`test.c`** para verificar las funciones en casos extremos:

- ¿Funciona bien `stack_pop()` para pilas de tamaño `1`?
- Si la pila queda vacía, *¿puedo volver a insertar elementos?*

- ¿La función `stack_to_array()` devuelve `NULL` para una pila vacía? *¿Devuelve los elementos en el orden correcto?*

Como aplicación del TAD, implementar en la función `main()` del archivo **`reverse.c`** un algoritmo que utilice el TAD Stack para invertir un arreglo de enteros leído desde un archivo. Se espera que el programa funcione por línea de comandos de la siguiente manera:

```
$ ./reverse input/example-easy.in
Original: [1, 2, 3, 4, 5]
Reversed: [5, 4, 3, 2, 1]
```

El algoritmo se puede describir como sigue:

1. Crear una pila vacía
2. Iterar sobre el arreglo de izquierda a derecha, insertando los números uno por uno en la pila.
3. Extraer cada uno de los elementos de la pila, usando `stack_top()` para obtenerlos y `stack_pop()` para removerlos.
4. Construir un arreglo nuevo con los elementos obtenidos.

Se incluye en la carpeta **`reverse`** un **`Makefile`**, por lo cual podrán compilar estando dentro de esa carpeta haciendo directamente:

```
$ make
```

b) Modificar la implementación del TAD Stack para que `stack_size()` sea de orden constante. Revisar si con esta nueva implementación hay una *invariante de representación* no trivial que puedan chequear con `assert()` en las operaciones. El comando **`reverse`** implementado en el apartado anterior debe seguir funcionando sin cambios para esta nueva versión del TAD.

Ejercicio 2: Stack con arreglos

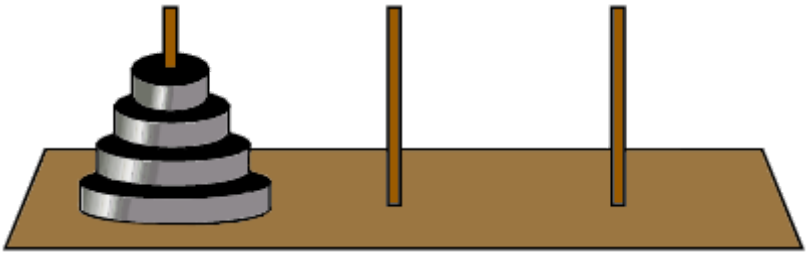
Hacer una nueva implementación del TAD Stack usando **arreglos dinámicos**. La pila debe ser capaz de almacenar cualquier cantidad de elementos. Internamente se deben almacenar los elementos en un arreglo. La estructura de representación está dada por:

```
struct _s_stack {
    stack_elem *elems;      // Arreglo de elementos
    unsigned int size;      // Cantidad de elementos en la pila
    unsigned int capacity;  // Capacidad actual del arreglo elems
};
```

Como se puede ver el campo `elems` es un puntero que cumplirá el rol de apuntar a un arreglo que puede alojar como máximo `capacity` elementos. Por otro lado `size` indicará la cantidad de elementos que se encuentran efectivamente en la pila. Cuando se intente agregar un elemento y el arreglo se encuentre lleno (`size == capacity`) debe pedirse más memoria para `elems` usando la función `realloc()`. Para no hacer demasiadas realocaciones de memoria, no conviene pedir espacio solo para el nuevo elemento sino que es recomendable pedir para el doble de la capacidad actual. No es necesario llamar a `realloc()` en las llamadas a `stack_pop()` (no vamos a achicar el arreglo). Asegúrense de verificar la *invariante de representación* para esta implementación.

Ejercicio 3: Torres de Hanoi

En el clásico juego de las torres de Hanoi, se tienen tres torres y N discos de distintos tamaños, que inicialmente se encuentran como lo muestra la siguiente imagen:



Se desean mover los `N` discos a la tercera torre, usando la torre del medio como auxiliar. Sólo es posible mover un disco a la vez, y nunca se debe apoyar un disco grande sobre uno más pequeño. Ver wikipedia para una descripción más completa del juego y su historia.

El argumento del programa es el número de discos a utilizar.

```
$ ./solve-hanoi 3
```

En la implementación que se encuentra en **`hanoi.c`**, cada torre se representa con una pila. Cada disco se representa con un número entero que indica su tamaño. Inicialmente usamos `hanoi_init()` para construir las tres pilas, la primera tiene como elementos la secuencia `[4,3,2,1]` (asumiendo *N=4* como en la imagen anterior) y el resto son pilas vacías.

La función `hanoi_solve()` muestra en la pantalla el estado de las torres luego de cada movimiento. El código implementado se basa en el siguiente pseudocódigo:

```
mover(N, Source, Target, Auxiliar):
  if N > 0:
    // Mover N – 1 discos de Source a Auxiliar
    mover(N – 1, Source, Auxiliar, Target)
    // Mover el disco que queda a Target
    Target.push(Source.top())
    Source.pop()
    // Mover N - 1 discos de Auxiliar a Target
    mover(N – 1, Auxiliar, Target, Source)
  endif
```

Se provee una función para “dibujar” las torres en consola, pero puede modificarse a gusto!

El ejercicio se trata de **solucionar un par de bugs** del programa y **verificar *leaks* de memoria**.

Se debería detectar el bug usando las implementaciones del TAD Stack de los ejercicios *1b)* y *2)*, pero no debería haber problemas al usar la implementación del *1a) ¿Por qué será esto?*

Para verificar que el programa no tenga *memory leaks* se debe usar la herramienta **`valgrind`**:

```
$ valgrind --leak-check=full ./solve-hanoi 10
```

El reporte de la herramienta debe mostrar que no hay bytes perdidos de memoria. Ejemplo:

```
==7929== HEAP SUMMARY:
==7929==     in use at exit: 0 bytes in 0 blocks
==7929==   total heap usage: 244 allocs, 244 frees, 1,928 bytes
allocated
==7929== All heap blocks were freed -- no leaks are possible
```

También se deben corregir los errores (Invalid reads/writes) si los hubiera. Para este ejercicio se incluye un **`Makefile`** por lo cual la compilación se realiza de la siguiente manera:

```
$ make
```

Se puede agregar una regla en **`Makefile`** para ejecutar su programa usando **`valgrind`**.

Se recomienda **primero eliminar los *leaks* de las implementaciones de Stack** y luego revisar las de **`solve-hanoi`**.