Algoritmos y Estructuras de Datos II

TALLER - 9 de mayo de 2024

Laboratorio 5: Stack

- Revisión 2024: Marco Rocchietti

Objetivos

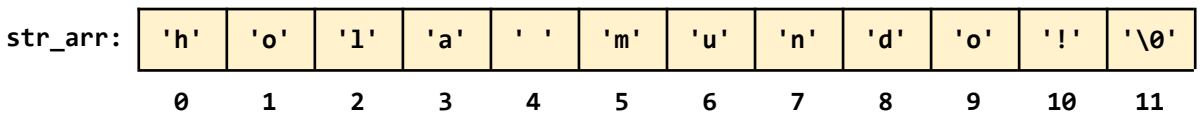
1. Trabajar con cadenas en C como **`char*`**
2. Introducir redirección de entrada estándar
3. Implementa el TAD Pila con distintas representaciones
4. Concepto y verificación de invariantes de representación
5. Creación de arreglos dinámicos
6. Funciones `calloc()` y `realloc()`
7. Usar **`valgrind`** para detectar problemas de manejo de memoria
8. Usar **`gdb`** para ayudar a solucionar *bugs* de programación

Preliminares: Cadenas y Memoria

Como se vio en el Lab02, las cadenas en C se pueden implementar como arreglos de caracteres, por ejemplo en

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

el contenido del *array* es el siguiente:

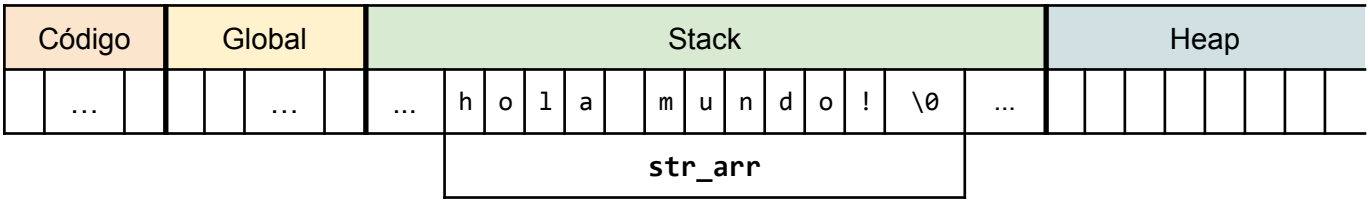


En realidad cuando se requiere usar cadenas en C se espera un tipo **`char*`** o sea un puntero a tipo **`char`**. En otras palabras una cadena es una secuencia de valores de tipo **`char`** que termina ante la aparición de un caracter **`'\0'`** y para acceder a la cadena se necesita un puntero (dirección de memoria) al primer caracter. En el ejemplo de arriba todo funciona bién porque cuando escribimos el nombre de un arreglo sin indexarlo obtenemos la dirección de memoria del primer elemento, por lo que la expresión **`str_arr`** es del tipo **`char*`** y entonces **`printf()`** recibe el tipo esperado para cadenas.

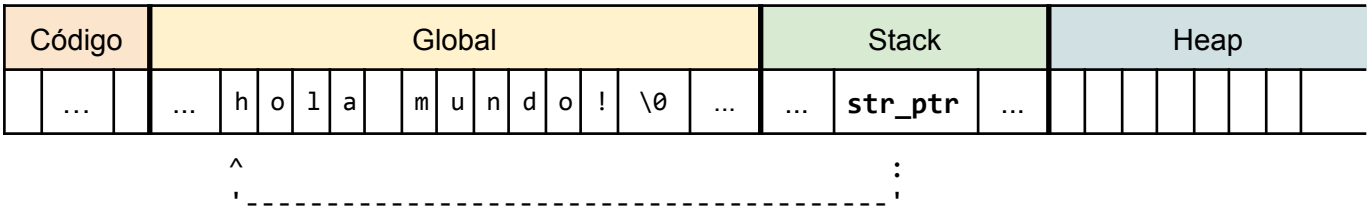
Una forma de hacer lo mismo de arriba pero con punteros es la siguiente:

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

Aunque no parece, hay grandes diferencias entre ambas alternativas. Para el arreglo **str_arr** lo que sucede es que en tiempo de compilación se determina el tamaño necesario para alojar los símbolos de **"hola mundo!"** y se establece el tamaño del arreglo en ese valor (en este caso 12 elementos de tipo **char**), luego se inicializa cada una de las posiciones con el símbolo correspondiente. Cuando el programa se ejecute, el arreglo se va a alojar en la sección del Stack de la memoria ya que es un arreglo estático. Entonces ubicándolo en el esquema de memoria que se vio anteriormente:



Por otro lado para la definición de **str_ptr** lo que sucede es distinto: el compilador al encontrarse con la cadena **"hola mundo!"** la definirá como datos globales del programa, reservándole un lugar en la sección Global de la memoria. Luego a **str_ptr** le asigna la dirección de memoria dónde están estos datos, es decir **str_ptr** apunta a la cadena **"hola mundo!"** que está alojada en la sección global:



En consecuencia una de las diferencias más importantes es que mientras **str_arr** es un arreglo en el que se pueden modificar los elementos que contiene (e.g. se puede hacer **str_arr[0]='H';**), no es posible cambiar el contenido de lo que apunta **str_ptr** puesto que la memoria de las sección global es de solo lectura (probar hacer **str_ptr[0]='H';** y se podrá apreciar una contundente violación de segmento).

Otra diferencia (ya vista cuando se compararon arreglos vs punteros) es que se puede reutilizar **str_ptr** para que apunte a otro lugar, donde haya otra cadena:

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

Nuevamente la cadena **"hola somos campeones del mundo!"** también será alojada en la sección Global de sólo lectura. Notar que en el ejemplo no se cambia el contenido de la memoria apuntada por **str_ptr** sino que se está cambiando a donde apunta **str_ptr**. Es también importante darse cuenta que no es necesario liberar **str_ptr** ya que la memoria Global es inmutable durante toda la ejecución del programa, una vez que se carga el programa no se crean ni destruyen elementos. Obviamente si se cambia la dirección a donde apunta **str_ptr** a un lugar del Stack o del Heap, se puede usar **str_ptr** para modificar elementos. Por ejemplo:

```
char str_arr[]="hola mundo estático!";
char *str_ptr="hola mundo global!";
printf("cadena: %s\n", str_ptr);
str_ptr = str_arr;
str_ptr[0] = 'H';
printf("cadena: %s\n", str_ptr);
str_ptr = malloc(sizeof(char)*22); // Lugar para \0 y uno extra por "á"
strcpy(str_ptr, "hola mundo dinámico!");
str_ptr[0] = 'H';
printf("cadena: %s\n", str_ptr);
free(str_ptr);
str_ptr=NULL;
```

La salida será

```
cadena: hola mundo global!
cadena: Hola mundo estático!
cadena: Hola mundo dinámico!
```

En el ejemplo el puntero **str_ptr** navega por todos los tipos de memoria vistos. Particularmente con **strcpy()** se está copiando la cadena **"hola mundo dinámico!"** a la memoria apuntada por **str_ptr**.

Ejercicio 1: Cadenas

Las siguientes consignas deben resolverse sin utilizar funciones de las librerías estándar **<string.h>** ni **<strings.h>** (salvo en el apartado b). Pueden reutilizar algo del código hecho para el tipo **fixstring** en laboratorios anteriores. Responder las preguntas que se formulen en un comentario al final del código:

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**, 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**, y la función:

```
bool string_is_symmetric(const char *str);
```

que indica si la cadena apuntada por **str** es simétrica en cuanto que el primer caracter coincide con el último, el segundo con el penúltimo, etc; como por ejemplo las cadenas **"aba"**, **"abcba"**, **"a"**, **""**.

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)

La cadena 'abcba' resulta ser simétrica
La cadena 'ab' resulta NO ser simétrica
```

b) En el archivo **checkpal.c** se encuentra implementado un programa que busca verificar si un texto ingresado por teclado (más precisamente por la entrada estándar *stdin*) es o no un palíndromo (ver en wikipedia). El programa usa la librería **strfuncs** implementada anteriormente por lo que se debe copiar al directorio e incluirla en la compilación. Identificar los problemas generados por **scanf()** probando distintas entradas. Además en los archivos **pal1.in**, **pal2.in**, **pal3.in**, **pal4.in** y **pal5.in** hay ejemplos de palíndromos y en los archivos **nopal1.in**, **nopal2.in** y **nopal3.in** ejemplos de textos que no lo son. Un truco para evitar escribir por teclado el contenido de los archivos es ejecutar el programa redirigiendo su entrada estándar al archivo que se quiere probar:

```
$ ./checkpal < pal1.in
```

de esa manera **scanf()** (y cualquier otra función que lea la entrada estándar) leerá del archivo **pal1.in.** Reemplazar **scanf()** 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 **string_length()** de **strfuncs**. Por último resolver el problema de *memory leaks* usando valgrind:

```
$ valgrind --leak-check=full ./checpal
```

c) Analizar la función **string_clone()** y determinar cuál es el problema en su implementación. Se debe incluir la descripción del problema encontrado como un comentario al final del código. Si a simple vista no se identifica el error, utilizar **valgrind**:

```
$ valgrind --track-origins=yes ./clone
```

Usar **gdb** para averiguar el valor que toma el parámetro **length** cuando se llama a **string_clone()** (también se debe anotar como comentario). Modificarla para que funcione correctamente. Modificar la función **main()** para que no queden *memory leaks*. Una vez completada la función **string_clone()** se debe crear el archivo **clone_ptr.c** copiando el código de **clone.c**. Luego verificar qué sucede si se cambia en **clone_ptr.c** el tipo de la variable **original** por:

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

Corregir el problema y explicar en un comentario al final del código por qué no funciona de manera correcta el código con el cambio de tipo.

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>`** como **`strcpy()`**, **`strlen()`**, etc. 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 2: 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 3: Stack con arreglos

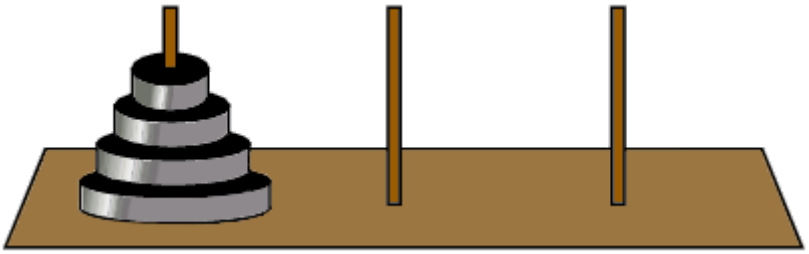
Hacer una nueva implementación del TAD Stack usando **arreglos dinámicos**. La pila debe ser capaz de almacenar cualquier cantidad de elementos. Internamente se deben almacenar los elementos en un arreglo. La estructura de representación está dada por:

```
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 4: 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:

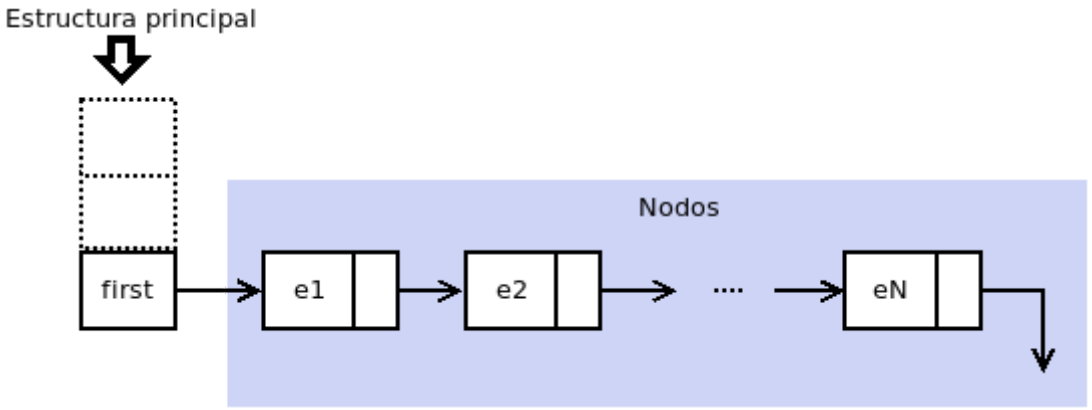
```
$ make
```

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`**.

Ejercicio 5: TAD Queue

Se debe completar la implementación del TAD Queue cuya representación consiste en una estructura principal que posee un campo **`first`** que apunta al primer nodo de la cola:



La línea punteada en el esquema indica que en la implementación final pueden haber más campos en la estructura principal. Se provee **`main.c`** que lee un archivo que se le pasa como parámetro desde la línea de comandos y carga los elementos a la cola. Un ejemplo de ejecución:

```
$ ./readqueue ../input/example_easy.in
length: 5
[ 1, 2, 3, 4, 5]
```

Se recomienda crear un **`Makefile`** para compilar. Asegurarse de usar las flags habituales: `-Wall -Werror -Wextra -pedantic -std=c99`. Sería conveniente que cada apartado tenga su propio **`Makefile`**

a) Completar la implementación del TAD Queue en **`queue.c`**. Agregar los campos necesarios a la estructura principal para que las función **`queue_size()`** sea de orden constante *O(1)*. Además se debe modificar **`main.c`** para que no haya *memory leaks.*

b) Implementar la nueva función:

```
queue queue_disscard(queue q, unsigned int n);
```

que elimina de la cola el n-ésimo elemento, siendo **`queue_disscard(q, 0)`** equivalente a **`queue_dequeue(q)`**. Agregar a **`main.c`** el código necesario para que el usuario pueda eliminar uno de los elementos de la cola. Para ello utilizar la función **`queue_user_disscard()`** que se declara en **`queue_helpers.h`** y ya viene implementada. Luego de que el usuario elimine el elemento, volver a mostrar el contenido de la cola.

c) Usar como base la implementación del apartado ***b)*** y aprovechar la estructura principal para lograr que tanto **`queue_enqueue()`** como **`queue_dequeue()`** sean *O(1)* (se puede agregar un puntero extra para ello). Copiar **`main.c`** del apartado ***a)*** y comparar la ejecución de las implementaciones hechas en ***a)*** y ***c)*** leyendo los archivos **`unsorted-100000.in`** y **`unsorted-200000.in.`**