Algoritmos y Estructuras de Datos II

TALLER - 8 de mayo 2025

Laboratorio 4: Punteros y memoria dinámica

- Revisión 2024: Marco Rocchietti

- Revisión 2025: Franco Luque

Código

lab04-kickstart.tar.gz

Objetivos

1. Comenzar a trabajar con punteros en C
2. Simular variables de salida con punteros
3. Comprender uso de punteros para mayor desempeño
4. Administración de memoria dinámica (`malloc()`, `calloc()`, `free()`)

Primera parte: Punteros 101

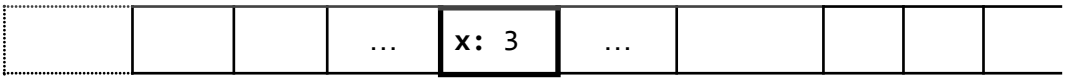
El objetivo del primer ejercicio es adquirir un entrenamiento básico y comprender el funcionamiento de punteros en C.

Un puntero es un tipo de variable especial que guarda una dirección de memoria. En C se denotan los punteros usando el símbolo `*` . Es decir que una variable `p` declarada como `int *p;` es del tipo *puntero a* `int`.

Para el manejo de punteros contamos con dos operadores unarios básicos, el operador de referenciación y el de desreferenciación.

Operación de referenciación (&)

Este operador obtiene la dirección de memoria de una variable. También se lo conoce como operador de dirección (*address operator*). Si se tiene una variable entera `x` declarada como `int x=3;` entonces la expresión `&x` retornará la dirección de memoria donde está alojado el contenido de la variable `x`.

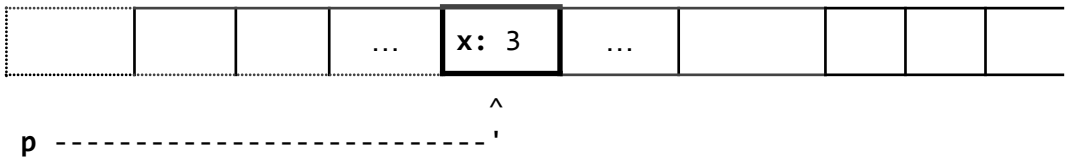


(memoria)

Particularmente la expresión `&x` en este caso es del tipo `int*`, es decir del tipo puntero a `int`. Por lo tanto, se puede hacer lo siguiente:

```
int x=3;
int *p;
p = &x;
```

notar que la asignación es correcta porque **p** y **&x** tienen el mismo tipo. En este ejemplo entonces el puntero **p** guarda la dirección de memoria de **x** y podemos decir que ***p** apunta a **x**.*



Operación de desreferenciación (*)

Obtiene el **valor** de lo *apuntado* por el puntero. También se lo conoce como el operador de indirección (*indirection operator*). Se lo puede pensar como una operación de inspección ya que accede al valor alojado en una dirección de memoria. Si se tiene una variable de tipo **int*** llamada **p**, entonces la expresión ***p** retornará el valor entero que se aloja en la dirección de memoria que contiene **p**. En el ejemplo de más arriba ***p** devuelve **3**.

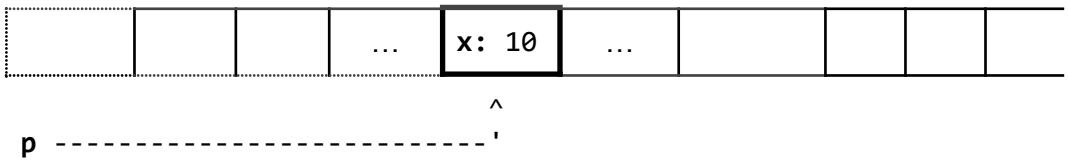
Además si se utiliza ***p** del lado izquierdo de una asignación:

```
*p = <expresión>;
```

la asignación escribirá el resultado de la expresión en la dirección de memoria apuntada por **p**, por lo que se cambia el valor contenido en esa dirección de memoria (sin modificar a **p**, que seguirá apuntando al mismo lugar). Por ejemplo, si se hace la asignación

```
int x=3;
int *p;
p = &x;
*p = 10;
```

entonces sucede que



Notar que se cambió el valor de la variable **x** de manera indirecta usando el puntero **p**.

Cabe aclarar que cuando se declara la variable de tipo puntero **int *p;** el símbolo ***** no actúa como operador sino que simplemente indica que la variable **p** se declara como puntero.

Para pensar: ¿Qué valor tendrá la variable **y** luego de ejecutar el siguiente código?

```
int x = 3;
int y = 10;
```

```
y = *(&x);
```

En el *Laboratorio 1* se utilizaba la función **`fscanf()`** ¿Qué parámetros tomaba dicha función y cuál era el tipo de cada uno?

Constante nula de punteros (`NULL`)

Siempre es buena idea dar un valor inicial a las variables apenas se declaran. Para punteros existe en C la constante `NULL` que representa una dirección de memoria nula, en la cual no se puede leer ni escribir. Esta constante es una macro definida en los *headers* de **`stdlib.h`** como la dirección de memoria `0`.

Recordar que si no se inicializa una variable esta puede contener cualquier valor, en el caso de enteros podría ser un número muy grande y extraño (o quizás un inofensivo 4) y en el caso de punteros puede tener asignada una dirección de memoria que en caso de querer escribir o leer de ella generaría problemas. Por ejemplo en el siguiente programa:

```
int *p;
*p = 3;
```

es fácil imaginar que esto podría generar que el programa termine con un error (violación de segmento - *segmentation fault*) pero podría ser peor. Puede suceder que por azar en **`p`** se encuentre la dirección de memoria de otra variable del programa y la modifiquemos, generando un **BUG** muy difícil de rastrear.

En esta otra versión:

```
int *p=NULL;
*p = 3;
```

el programa siempre va a fallar, y eso es bueno.

Claramente los ejemplos de arriba son errores que saltan a la vista pero sirven para ilustrar situaciones en la que no es tan obvio que se usa un puntero sin inicializar, pero el efecto es el mismo.

Operadores de indexación y flecha

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 (de manera alineada al tipo de datos del puntero) 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`**).

Ejercicio 1: Introducción de punteros

La tarea de este ejercicio consiste en completar el archivo **`main.c`** de manera tal que la salida del programa por pantalla sea la siguiente:

```
x = 9
m = (100, F)
a[1] = 42
```

Las restricciones son:

- No usar las variables **`x`**, **`m`** y **`a`** en la parte izquierda de alguna asignación.
- Se pueden agregar líneas de código, pero no modificar las que ya existen.
- Se pueden declarar hasta 2 punteros.

Recordar siempre inicializar los punteros en **`NULL`**:

Se mostró en el taller cómo hacer debugging de un programa mediante GDB. Esta herramienta también es útil para entender "qué está pasando" con el código cuando se ejecuta. Se recomienda compilar con los símbolos de debugging y poner breakpoints para imprimir los valores de las variables del programa.

En gdb también se pueden imprimir valores como:

- *Dirección de memoria de una variable:* **`print &x`**
- *El valor que hay en la memoria apuntada por un puntero:* **`print *p`**

Ejercicio 2: Simulando parámetros **out** de procedimientos

En el lenguaje de programación del teórico-práctico se usan funciones y procedimientos que tienen una naturaleza distinta a las funciones de C. Particularmente en C sólo existen las funciones y a veces llamamos "procedimientos" a aquellas funciones que devuelven algo del tipo **`void`** (que es el tipo "vacío" de C, o en otras palabras que no devuelve nada). Se debe entonces traducir el siguiente programa tratando de simular el procedimiento **`absolute()`** usando funciones de C :

```
proc absolute(in x : int, out y : int)
  if x >= 0 then
    y := x
  else
    y := -x
  fi
end proc

fun main() ret r : int
  var a : int
  a := -10
```

```
  absolute(a, res)
  {- supongamos que print() muestra el valor de una variable -}
  print(res)
  {- esta última asignación es análoga a `return EXIT_SUCCESS;` -}
  r := 0
end fun
```

¿Qué valor se mostraría al ejecutar la función **main()** del programa anterior?

a) Abrir el archivo **abs1.c** y traducir a lenguaje C usando el siguiente prototipo para **absolute()**:

```
void absolute(int x, int y) {
  (⋮)
}
```

luego compilar con el siguiente comando:

```
$ gcc -Wall -Werror -pedantic -std=c99 abs1.c -o abs1
```

(notar que no se utiliza **-Wextra** sólo por esta vez) y ejecutar

```
$ ./abs1
```

¿Qué valor se muestra por pantalla? ¿Coincide con el programa en el lenguaje del teórico? Responder en un comentario al final de **abs1.c**.

b) Ahora abrir el archivo **abs2.c** que utiliza el siguiente prototipo de **absolute()**:

```
void absolute(int x, int *y) {
  (⋮)
}
```

Pensar qué modificaciones son necesarias hacer dentro de las funciones **absolute()** y **main()** respecto a la implementación realizada en **abs1.c** para trabajar con el nuevo prototipo. Además se debe lograr que el programa en C simule el comportamiento del programa original en lenguaje del teórico-práctico. Implementar esas modificaciones en **abs2.c** y luego compilar:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 abs2.c -o abs2
```

y por último ejecutar

```
$ ./abs2
```

¿Se muestra el valor correcto? (en caso contrario revisar hasta lograr que sí lo haga)

❗	*Notar que conceptualmente en ningún caso los programas son equivalentes puesto que las funciones y procedimientos del lenguaje del teórico tienen una naturaleza completamente distinta a las de C.*

c) Para pensar:

- ¿El parámetro **`int *y`** de `absolute()` es de tipo **`in`**, de tipo **out** o de tipo **`in/out`**?

- ¿Qué tipo de parámetros tiene disponibles C para sus funciones?
  - Parámetros **`in`**
  - Parámetros **`out`**
  - Parámetros **`in/out`**

Responder al final de **`abs2.c`** como un comentario en el código

d) En un nuevo archivo **`swap.c`** implementar una traducción del programa que intercambia valores:

```
proc swap(in/out a : int, in/out b : int)
    var aux : int
    aux := a
    a := b
    b := aux
end proc

fun main() ret r : int
  var x, y : int
  x := 6
  y := 4
  print(x, y)
  swap(x, y)
  print(x, y)
  r := 0
end fun
```

Segunda parte: Memoria dinámica

Visualizando direcciones de memoria

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", (void *) p);
```

Notar el *casteo* que se le realiza a **`p`** cuando se lo pasa como parámetro a **`printf()`**. Cuando antes de una expresión se introduce un tipo entre paréntesis, significa que la expresión se va a convertir al tipo indicado. Por ejemplo **`(float) 2`** hace que el resultado se interprete como **`2.0f`**, otro ejemplo puede ser **`(int) 1.5`** que hace que el resultado sea el entero **`1`**. En este caso se convierte a **`p`**, que es un **`int*`**, a un **`void*`** o sea un puntero a **`void`** lo cual es simplemente una dirección de memoria pura, puesto que un puntero a **`void`** no se puede desreferenciar.

La salida del programa va a ser un número en hexadecimal (con prefijo `0x`…), por ejemplo:

```
La dirección de memoria apuntada por p es: 0x7ffd15a1ac60
```

Otra forma con la cual se puede ver el valor decimal de una dirección de memoria es hacer:

```
int *p=NULL;
int a=55;
p = &a;
printf("El índice de memoria alojado en p es: %lu", (uintptr_t) p);
```

aquí se castea el puntero a un entero sin signo que es el tipo **`uintptr_t`** definido en **`<stdint.h>`**. Este tipo es lo suficientemente grande para guardar direcciones de memoria, pero la solución no es muy estándar ya que el comodín **`"%lu"`** podría fallar en algunos sistemas ya que **`printf()`** espera algo del tipo **`long unsigned int`** y podría no ser equivalente a **`uintptr_t`**. La salida sería algo como:

```
El índice de memoria alojado en p es: 140724966370400
```

Memoria dinámica: Stack vs Heap

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()`**:

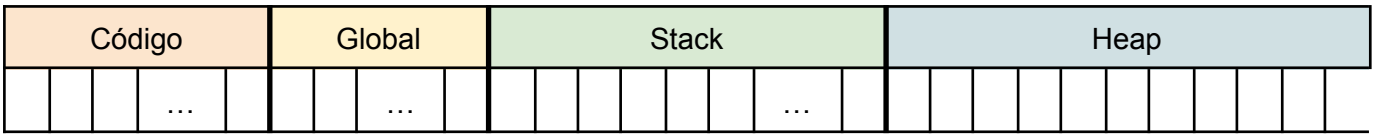
```
int *p=NULL;
p = malloc(sizeof(int));
*p = 5;
free(p);
```

La función **`malloc()`** toma un parámetro, que es un entero sin signo de tipo **`size_t`** (muy parecido a **`unsigned long int`**) que es la cantidad de memoria en *bytes* que se solicita reservar. A diferencia de **`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()`** (aún si uno hubiera memorizado cuantos *bytes* ocupa cada tipo en su computadora, esto puede variar según la versión del sistema operativo o el microprocesador en el que se use el programa).

```
$ man malloc
```

Las direcciones de memoria que devuelve **`malloc()`** se encuentran en la sección de memoria

denominada *Heap* (no confundir con la estructura de datos que lleva el mismo nombre ya que no tiene ninguna relación).



A modo informativo, el mapa de más arriba es un esquema de cómo se organiza la memoria. La sección de *Código* contiene las instrucciones del programa, la sección *Global* contiene las variables globales, la sección *Stack* es donde están las variables que usamos en las funciones de nuestro programa (memoria estática) y la sección *Heap* es la región de la memoria dinámica la cual se reserva y libera manualmente mediante `malloc()` y `free()`. El *Stack* por su parte se maneja de manera automática, reservando memoria para las variables declaradas en una función que se comienza a ejecutar y liberando esa memoria cuando la función termina su ejecución.

Otra gran diferencia entre el *Stack* y el *Heap*, es que la cantidad de memoria asignada para el Stack es limitada. Si los datos contenidos en el *Stack* superan dicho límite se genera un ***stack overflow***. Si durante la ejecución de un programa se está ejecutando una función `f1` y ésta llama a su vez a otra función `f2`, durante la ejecución de `f2` las variables de `f1` siguen en el *Stack* ya que aún no se terminó de ejecutar. Las variables de las funciones llamadas de manera anidada se van apilando entonces. Hay en consecuencia un límite en la cantidad de llamadas anidadas de funciones, particularmente un número máximo de llamadas recursivas. La cantidad dependerá de cuánta memoria ocupen las variables de las funciones involucradas. Esto hace que si una función declara un arreglo en memoria estática muy grande, podría dejar poco margen para llamadas a otras funciones o directamente generar un *stack overflow* porque el arreglo no entra en el *Stack*.

Por su parte la memoria en el *Heap* tiene disponible toda la memoria *RAM* de la computadora, por lo que mientras haya memoria libre se podrá pedir reservar nueva memoria mediante `malloc()`. Pero *un gran poder conlleva una gran responsabilidad*, por lo que no se debe olvidar liberar la memoria reservada cuando deje de usarse, puesto que los ***memory leaks*** pueden generar a la larga que la computadora se bloquee por completo.

Ejercicio 3: Sizeof, malloc y free

a)

1. 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?*
2. De manera similar a lo hecho para mostrar los tamaños de cada campo de la estructura, agregar un mensaje que muestre la dirección de memoria de cada campo. Se recomienda usar los dos tipos de visualizaciones explicadas anteriormente (direcciones e índices). Analizar la salida y responder: *¿Hay algo raro en las direcciones de memoria?*
3. Por último, modificar `sizes.c` para que la estructura data_t se guarde en la memoria dinámica. Usar `malloc` y `free` para reservar y liberar la memoria.

b) 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);
```

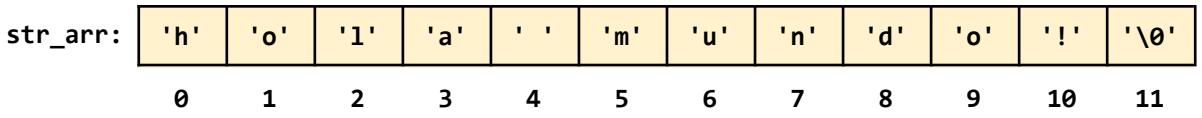
que carga los datos del archivo **`filepath`** devolviendo un puntero a memoria dinámica con los elementos del arreglo y dejando en **`*length`** la cantidad de elementos leídos. Completar además en **`main.c`** el código necesario para liberar la memoria utilizada por el arreglo. Probar el programa con todos los archivos de la carpeta **`input`** para asegurar el correcto funcionamiento (notar que la versión en **`static`** no funciona para todos los archivos de la carpeta **`input`**).

Tercera parte: 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:

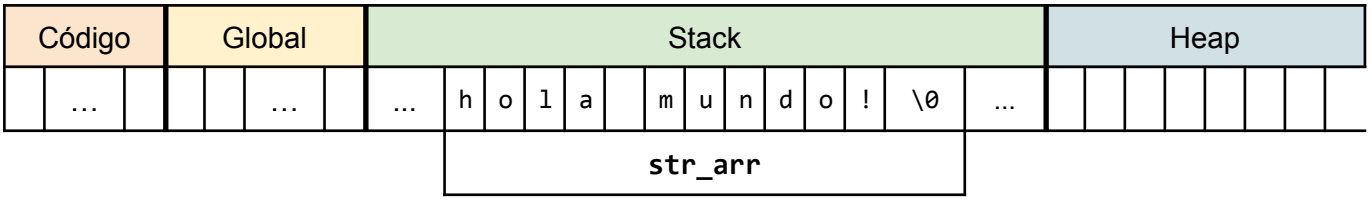


En realidad cuando se requiere usar cadenas en C se espera un tipo **`char*`** o sea un puntero a tipo **`char`**. En otras palabras una cadena es una secuencia de valores de tipo **`char`** que termina ante la aparición de un caracter **`'\0'`** y para acceder a la cadena se necesita un puntero (dirección 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.

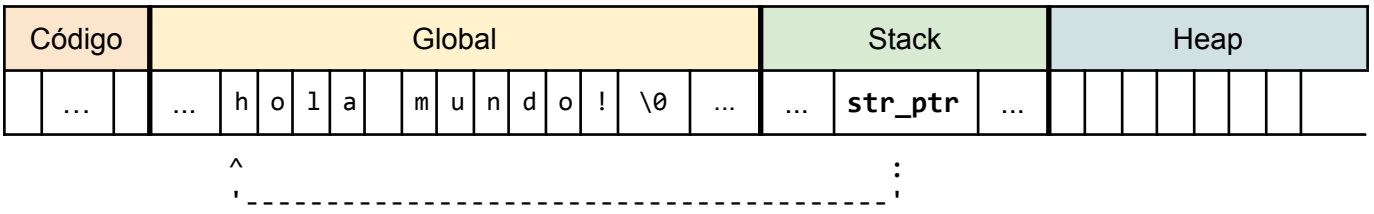
Una forma de hacer lo mismo de arriba pero con punteros es la siguiente:

```
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 4: 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()`**.