Algoritmos y Estructuras de Datos II

TALLER - 13 de abril 2023

Laboratorio 3: Matrices e Introducción a Punteros

- Revisión 2023: Marco Rocchietti

Objetivos

1. Comenzar a familiarizarse con punteros en C
2. Simular variables de salida con punteros
3. Ejercitar la resolución de problemas
4. Uso de arreglos multidimensionales y tipos **`enum`**
5. Uso de arreglos con elementos de tipo **`struct`**
6. Uso de redirección de **`stdout`** por línea de comandos
7. Lectura robusta de archivos
8. Comprender uso de punteros para mayor desempeño

Ejercicio 1: Introducción de punteros

El objetivo de este 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 representan 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:

- Referenciación (**&**): obtiene la dirección de memoria de una variable. Si se tiene una variable entera `x` declarada como **`int x;`** entonces la expresión **`&x`** retornará la dirección de memoria donde está alojado el contenido de la variable `x`. Particularmente la expresión **`&x`** en este caso es del tipo **`int *`** por lo que se podría hacer **`int *p=&x;`** ya que tipa**.** El operador tambien se lo conoce como operador de dirección (*address operator*)
- 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`**. 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 el valor de `p`, que seguirá teniendo la misma dirección). 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. A este operador 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.

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?

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`:

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

En *C* la constante `NULL` es una macro definida en los *headers* de **`stdlib.h`** como la dirección de memoria `0` que se utiliza para representar al puntero que no apunta a ningún lugar, también llamado nulo.

Ayuda: 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. También se pueden imprimir valores como:

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

Ejercicio 2: Simulando 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, b : 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?

Abrir el archivo **`proc1.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 proc1.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 **`proc1.c`**.

Ahora abrir el archivo **`proc2.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 **`proc1.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 **`proc2.c`** y luego compilar:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 proc2.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.*

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 **`proc2.c`** como un comentario en el código

Ejercicio 3: Lectura con patrones

Parte A: Parseo de entrada

Se debe implementar un programa que cargue los datos de un archivo (cuya ubicación se pasa como parámetro) donde su contenido sigue el siguiente formato:

```
<int> -> *<char>*
<int> -> *<char>*
<int> -> *<char>*
(:)
```

el formato descripto espera en `<int>` un valor entero y en `<char>` un caracter. Notar que además se espera que el caracter de entrada esté entre medio de dos símbolos asterisco (`*`), también se espera que luego del entero haya una flecha compuesta por el símbolo `-` y luego `>`. Un ejemplo concreto del formato es el del archivo **`phrase-basic.in`** cuyo contenido es:

```
2 -> *l*
0 -> *h*
3 -> *a*
4 -> *!*
1 -> *o*
```

Notar que a diferencia de los formatos utilizados anteriormente, aquí no hay información de cuántos

elementos tiene el archivo. Por ello para saber cuándo se debe dejar de leer elementos será necesario usar la función `feof()` definida en **`stdio.h`**. Para ver la documentación se puede consultar el manual de linux: **`man feof`**.

La función **`fscanf()`** *automáticamente saltea los espacios y caracteres* **`\n`** *hasta encontrar los datos indicados por el patrón (o falla al encontrar un dato que no respete el patrón indicado). Luego de leer los datos, el cursor de lectura se posiciona justo después de los datos leídos. Se aconseja poner* **`\n`** *al final del patrón o un espacio extra para asegurar que también se consuman los avances de línea y/o espacios remanentes posteriores a los datos leídos. De esta manera se evita que la última llamada de* **`fscanf()`** *devuelva* **`EOF`** *culpa de que el cursor haya quedado apuntando a uno de estos caracteres que no se consumió en la llamada previa (ya que en ese caso* **`feof()`** *va a indicar que todavía no se llegó al final del archivo, pero la próxima llamada a* **`fscanf()`** *fallará porque luego de los espacios y/o* **`\n`** *termina el archivo).*

Los datos leídos se deben almacenar en dos arreglos, uno para los valores enteros **`int indexes[]`** y otro para los caracteres **`char letters[]`**. Se recomienda definir una función

```
unsigned int
data_from_file(const char *path,
               unsigned int indexes[],
               char letters[],
               unsigned int max_size);
```

que además de llenar los arreglos con los datos leidos del archivo indicado por `path`, devuelva cuántos elementos efectivamente contenía dicho archivo. Notar que con `max_size` se indica la máxima cantidad de elementos que pueden almacenar los arreglos **`indexes[]`** y **`letters[]`**

No olvidar verificar los casos borde en la lectura de datos. Evitar escribir más elementos de los que se pueden almacenar en **`indexes[]`** *y* **`letters[]`***, asegurar la correcta lectura de los valores, etc…*

Para probar esta parte sin completar la *Parte B*, se puede modificar temporalmente en **`main.c`** la linea **`dump(sorted, length);`** por **`dump(letters, length);`** para que se muestren los datos cargados.

Parte B: Reconstrucción

Los archivos guardan las letras de un texto que se puede reconstruir ubicando cada letra en el índice especificado. En el arreglo **`sorted[]`** de **`main.c`** se debe dejar dicha reconstrucción. Entonces para el ejemplo **`phrase-basic.in`** se puede ver que el texto dice **`"hola!"`**. Se debe construir un programa que funcione de la siguiente manera:

```
$ ./readprhase phrase-basic.in
"hola!"
```

No se debe utilizar un algoritmo de ordenación para reconstruir la frase (hacer eso solo complicaría las cosas). Debe funcionar también para el resto de los archivos de ejemplo **`phrase1.in`**, …, **`phrase4.in.`** Pensar qué problemas pueden ocurrir si los índices que tiene el archivo son más grandes de lo previsto y tratar de evitar que el programa genere violaciones de segmento en esos casos.

En **`main.c`** se encuentra un esqueleto del ejercicio con una única función auxiliar implementada `dump()` que muestra el contenido de un arreglo de caracteres por pantalla. Se pueden agregar tantas funciones como se considere necesario y también agregar módulos si contribuye a mejorar la calidad del código. Para el manejo de parámetros de la función `main()` se puede reutilizar el código de laboratorios anteriores.

Ejercicio 4: Arreglos Multidimensionales

En el directorio del ejercicio se encuentran los siguientes archivos:

Archivo	*Descripción*
`main.c`	Contiene la función principal del programa
`weather.h`	Declaraciones relativas a la estructura de los datos climáticos y de funciones de carga y escritura de datos.
`weather.c`	Implementaciones **incompletas** de las funciones
`array_helpers.h`	Declaraciones / prototipos de las funciones que manejan la tabla del clima
`array_helpers.c`	Implementaciones **incompletas** de las funciones que manejan el arreglo

Parte A: Carga de datos

Abrir el archivo **`../input/weather_cordoba.in`** para ver cómo se estructuran los datos climáticos. Cada línea contiene las mediciones realizadas en un día. Las **primeras tres columnas** corresponden al *año*, *mes* y *día* de las mediciones. Las **restantes seis** columnas son la *temperatura media*, la *máxima*, la *mínima*, la *presión atmosférica*, la *humedad* y las *precipitaciones* medidas ese día.

Las temperaturas se midieron en grados centígrados (ºC) pero para evitar los números reales los grados están expresados en décimas (e.g. 15.2ºC está representado por 152 décimas). La presión (medida en *hectopascales*) también ha sido multiplicada por 10 y las precipitaciones por 100 (o sea que están expresadas en centésimas de milímetro). Esto permite representar todos los datos con números enteros. Cabe aclarar que para completar el ejercicio **no es necesario multiplicar ni dividir estos valores**, esta información es sólo para ayudar a la comprensión de los datos que se manejan.

La primera tarea consiste en completar el procedimiento de carga de datos en el archivo **`array_helpers.c`**. Recordar que el programa tiene que ser *robusto*, es decir, debe tener un comportamiento bien definido para los casos en que la entrada no tenga el formato esperado. Como guía se puede revisar el archivo **`array_helpers.c`** provisto por la cátedra en el *laboratorio 2*.

Una vez completada la lectura de datos se puede verificar si la carga funciona compilando,

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

y luego ejecutar

```
$ ./weather ../input/weather_cordoba.in > weather_cordoba.out
```

En la línea anterior, **`../input/weather_cordoba.in`** es el parámetro que se le pasa a nuestro programa **`weather`** (el archivo a procesar) y la parte **`> weather_cordoba.out`** hace que la salida del programa, en vez de mostrarse por la consola, se escriba en el archivo **`weather_cordoba.out.`** El archivo de salida será creado cuando comience la ejecución del programa (si **`weather_cordoba.out`** ya existía va a ser reemplazado).

Si no hubo ningún error, ahora se puede comparar la entrada con la salida:

```
$ diff ../input/weather_cordoba.in weather_cordoba.out
```

El programa **`diff`** (que ya viene instalado en linux) realiza una comparación de ambos archivos y sólo muestra las líneas que difieren. Si esto último no arroja ninguna diferencia, significa que tu carga funciona correctamente.

Si se compila el código original sin hacer modificaciones usando el *flag* **`-Werror`**, no compilará (ver explicación en *Ejercicio 2*). A modo de prueba, compilar el código original sin usar ese *flag* y ejecutar el programa resultante. *¿Qué se puede observar en la salida del programa?*

Parte B: Análisis de los datos

Construir una librería **`weather_utils`** que conste de los siguientes archivos:

- **`weather_utils.c`**
- **`weather_utils.h`**

La librería debe proveer tres funciones:

1. Una función que obtenga la menor temperatura mínima histórica registrada en la ciudad de Córdoba según los datos del arreglo.
2. Un “procedimiento” que registre para cada año entre 1980 y 2016 la mayor temperatura máxima registrada durante ese año.

   

   El procedimiento debe tomar como parámetro un arreglo que almacenará los resultados obtenidos.
3. Implementar un procedimiento que registre para cada año entre 1980 y 2016 el mes de ese año en que se registró la mayor cantidad mensual de precipitaciones.

Para el procedimiento del *ítem 2* se debería hacer algo parecido a lo siguiente:

```
void procedimiento(WeatherTable a, int output[YEARS]) {
   ⋮
  for (unsigned int year = 0; year < YEARS; year++) {
     ⋮
    output[year] = ... // la mayor temperatura máxima del año  'year' + 1980
     ⋮
  }
}
```

Finalmente modificar el archivo **`main.c`** para que se muestre los resultados de todas las funciones que se programaron.

Ejercicio 5: Ordenación de un arreglo de estructuras

En el directorio del ejercicio se encuentran los siguientes archivos:

Archivo	*Descripción*
`main.c`	Contiene la función principal del programa
`player.h`	Definición de la estructura para jugadores y definición de constantes
`helpers.h`	Declaraciones / prototipos de las funciones que manejan el arreglo de jugadores
`helpers.c`	Implementaciones de las funciones que manejan el arreglo
`sort.h`	Declaraciones / prototipos de las funciones relativas a la tarea de ordenación
`sort.c`	Implementaciones **incompletas** de las funciones de ordenación

Abrir el archivo **`../input/atp-players2022.in`** para visualizar los datos. Es un listado por orden alfabético de jugadores profesionales de tenis, actualizado a la semana del 11 de abril del 2022. En el contenido del archivo hay **seis columnas**: primero el *nombre del jugador*, luego una *abreviatura de su país*, un número que indica *el puesto que ocupa en el ranking*, su *edad*, su *puntaje* y el *número de torneos jugados* en el último año.

Al igual que en el *Laboratorio 2* hay un módulo que se encarga de manejar los arreglos y otro que maneja las funciones relativas a la ordenación. Se puede observar en las descripciones de los módulos qué cambios debieron hacerse entre **`helpers.h`** (de este lab) y **`array_helpers.h`** (del lab2) y qué cambios se hicieron entre **`sort.h`** (de este lab) y **`sort_helpers.h`** (del lab2) para adaptarse al nuevo tipo de arreglo (ya que en el laboratorio anterior se utilizaban arreglos de enteros).

Para compilar el ejercicio, primero se debe ejecutar

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c helpers.c sort.c
```

Si se compila el código original sin modificaciones, no compilará. ¿Por qué?. Si bien aparecen algunos errores por pantalla, estos son en realidad *warnings* del compilador. El compilador “advierte” que hay situaciones en el código que podrán llevar a errores de codificación. En este caso en particular dichos errores **se sabe** que corresponden a funciones incompletas en el código, entonces **sólo en este caso** se puede desactivar el *flag* **`-Werror`** que hace que los *warnings* de compilación sean tratados como errores.

Entonces, hasta que terminar de implementar los algoritmos de *sorting* se puede compilar:

```
$ gcc -Wall -Wextra -pedantic -std=c99 -c helpers.c sort.c main.c
$ gcc -Wall -Wextra -pedantic -std=c99 helpers.o sort.o main.o -o mysort
$ ./mysort ../input/atpplayers.in
```

No olvidar volver a poner la flag `-Werror` *al momento de compilar el código definitivo.*

Ahora para comprobar que la salida es idéntica a la entrada (salvo por la información sobre el tiempo utilizado para ordenar) se puede hacer:

```
$ ./mysort ../input/atpplayers.in > atpplayers.out
$ diff ../input/atpplayers.in atpplayers.out
```

Este ejercicio consiste entonces en realizar los cambios necesarios en el archivo **`sort.c`** para ordenar el arreglo cargado, de modo que el listado de salida esté ordenado según el puesto que el jugador tiene en el *ranking*. Se puede reutilizar el código del laboratorio anterior realizando las modificaciones que se consideren pertinentes y utilizar aquí cualquiera de los algoritmos de ordenación vistos en clase: *insertion sort*, *selection sort*, *quick sort*, etc.

Ejercicio 6: Aprovechando punteros para eficiencia

La intención del ejercicio es explorar la conveniencia de utilizar punteros para que los intercambios (*swaps*) sean más eficientes.

Completar el archivo **`sort.c`** copiando código del *Ejercicio 5* y realizando las modificaciones pertinentes para trabajar con arreglos de punteros a estructuras. Van a notar que en la nueva versión de **`player.h`** se redefinió al tipo `player_t`,

```
typedef struct _player_t {
      char name[100];
      char country[4];
      unsigned int rank;
      unsigned int age;
      unsigned int points;
      unsigned int tournaments;
} * player_t;
```

siendo entonces ahora **un puntero** a una estructura `struct _player_t`.

Notar que la función `main()` muestra la cantidad de tiempo empleado en la ordenación.

¿Funciona más rápido la versión con punteros? ¿Por qué son más eficientes los intercambios con esta versión?

La última línea de la función `main()` antes del `return` llama a `destroy()` *¿Por qué? ¿Qué ocurriría si esa línea no estuviera ahí?*