Algoritmos y Estructuras de Datos II

TALLER - 13 de abril 2021

Laboratorio 3: Matrices e Introducción a Punteros

- Revisión 2021: Marco Rocchietti
- Original 2019: Leandro Ramos

Objetivos

1. Parsear visualmente el símbolo de *prompt* ($)
2. Uso de arreglos multidimensionales
3. Uso de arreglos con elementos de tipo `struct`
4. Lectura robusta de archivos
5. Comenzar a familiarizarse con punteros en C
6. Simular variables de salida con punteros
7. Comprender uso de punteros para mayor desempeño

Ejercicio 1: Arreglos Multidimensionales

Parte A: Carga de datos

Abrí el archivo **`../input/weather_cordoba.in`** para ver cómo vienen 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 **no necesitás multiplicar ni dividir estos valores**, esta información es sólo para que puedas entender los datos.

Lo primero que tenés que hacer es completar el procedimiento de carga de datos en el archivo **`array_helpers.c`**. Recordá que el programa tiene que ser *robusto*, debiendo tener un comportamiento bien definido para los casos en que la entrada no tenga el formato esperado. Como guía podes revisar el archivo **`array_helpers.c`** del *Laboratorio 2*.

Una vez resuelta la lectura de datos podés verificar si la carga funciona compilando,

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

y luego ejecutando

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

Si compilás el código como te lo entregamos, usando el *flag* **`-Werror`** no compilará (ver explicación en *Ejercicio 2*). Compilá sin usar ese *flag* y ejecutá el código como está. *¿Qué podés observar en la salida del programa?*

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

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

Si esto último no arroja ninguna diferencia, significa que tu carga funciona correctamente.

Parte B: Análisis de los datos

Vas a hacer una librería **`weather_utils`** que consta de los siguientes archivos que debes crear:

- **`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.

   > **Ayuda:** 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* deberías 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 modifica el archivo **`main.c`** para que se muestre los resultados de todas las funciones que programaste.

Ejercicio 2: Ordenación de un arreglo de estructuras

En el directorio del ejercicio vas a encontrar 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

Abrí el archivo **`../input/atp-players2021.in`** para visualizar los datos. Es un listado por orden alfabético de jugadores profesionales de tenis actualizado a la semana del 12 de abril del 2021. Verás en el contenido del archivo **seis columnas**, donde el *nombre del jugador* viene acompañado de 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* tenemos un módulo que se encarga de manejar los arreglos y otro que maneja las funciones relativas a la ordenación. Podés 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 usábamos arreglos de enteros).

Para compilar el ejercicio, primero debes ejecutar

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

Si lo hacés con el código cómo te lo damos originalmente no compilará. ¿Por qué?. Si bien aparecen algunos errores por pantalla, vemos que 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 **sabemos** que corresponden a que hay funciones sin terminar, entonces **sólo en este caso** podemos desactivar el *flag* **`-Werror`** que hace que los *warnings* de compilación sean tratados como errores.

Entonces, hasta que terminemos de implementar los algoritmos de *sorting*:

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

Ahora el código compila, ¿verdad?

RECORDATORIO: No olvides volver a poner la *flag* **`-Werror`** al momento de compilar tu código definitivo. Tené en cuenta que al momento del parcial los profes compilaremos usando esta *flag* y si el código no compila de esta manera te **restará puntos**.

Ahora podés comprobar que la salida es idéntica a la entrada (salvo por la información sobre el tiempo utilizado para ordenar). Para comprobar eso, hacé

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

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

Ejercicio 3: Uso básico 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 declarada como `int *` es del tipo ***puntero a*** `int`.

Para el manejo de punteros contamos con dos operadores unarios básicos:

- Desreferenciación (*****): obtiene el **valor** de lo *apuntado* por el puntero. Supongamos que tenemos una variable de tipo `int*` llamada `p`, entonces `*p` retornará el valor entero que se aloja en dirección de memoria `p`.
- Referenciación (**&**): obtiene la dirección de memoria de una variable. Supongamos que tenemos una variable entera `x` declarada como `int x;` entonces `&x` retornará la dirección de memoria que aloja el valor contenido en la variable x.

Para pensar, ¿Qué valor tendrá `y` luego de ejecutar el siguiente código?

```
int x = 3;
int y = 10;
y = *(&x);
```

¿Recordás en el *Laboratorio 1* cuando hablamos de funciones como `fscanf()`? ¿Qué parámetros tomaba dicha función?

La tarea de este ejercicio consiste en completar el archivo **`main.c`** de manera 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.

Recuerda 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 `0` que se utiliza para representar al puntero que no apunta a ningún lugar, también llamado nulo.

Ayuda: La clase pasada vimos como hacer *debugging* de un programa con GDB. Esta herramienta también es útil para entender "qué está pasando" con mi código cuando se ejecuta. Intentá compilar con símbolos de *debugging* y poner *breakpoints* para imprimir los valores de las variables. También podés imprimir valores como:

- Tamaño 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 4: 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). Vamos entonces a simular el siguiente procedimiento visto en la primer clase del teórico-práctico:

```
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, res : int
  res := 0
  a := -98
  absolute(a, res)
  {- supongamos que print() muestra el valor de una variable -}
  print(res)
  r := 0
end fun
```

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

Abrí el archivo **proc1.c** y traducí a lenguaje *C* usando el siguiente prototipo para `abs()`:

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

luego compila haciendo

```
$ gcc -Wall -Werror -pedantic -std=c99 proc1.c -o abs1
```

(notar que no usamos **-Wextra** sólo por esta vez) y ejecuta

```
$ ./abs1
```

¿Qué valor se muestra por pantalla? ¿Coincide con el programa en el lenguaje del teórico?

Ahora abrí el archivo **proc2.c** que utiliza el siguiente prototipo de `abs()`:

```
void abs(int x, int *y) {
  (⁝)
}
```

Pensá qué modificaciones son necesarias hacer dentro de las funciones `abs()` y `main()` respecto a lo que habías hecho en **proc1.c** para trabajar con el nuevo tipo del segundo parámetro de `abs()` de manera tal que el programa en *C* simule el comportamiento del programa en el lenguaje del teórico. Implementá esas modificaciones en **proc2.c** y luego compilá haciendo

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 proc2.c -o abs2
```

y ejecutá

```
$ ./abs2
```

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

ATENCIÓN: 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, ¿Qué tipo de parámetros tiene C para sus funciones?

- Parámetros in
- Parámetros out
- Parámetros in/out

Ejercicio 5: Aprovechando punteros para eficiencia

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

Completá el archivo **`sort.c`** copiando código del *Ejercicio 2* 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?*