Sesión 09: Más sobre tipos y sintaxis (*a mejorar*)

Hoja de problemas

Programación para Sistemas

Ángel Herranz
aherranz@fi.upm.es
Universidad Politécnica de Madrid

2020-2021

Ejercicio 1. Repasa las transparencias de clase. Además de explorar los nuevos elementos (**`enum`**, **`union`**, y **`typedef`**), el aspecto más importante es la combinación de tantas cosas vistas en la sesiones anteriores y la exploración a fondo de ciertas partes de la sintaxis de C.

Ejercicio 2. Tu primera tarea va a consistir en implementar un programa que lea figuras geométricas de la entrada estándar e imprima el área de cada una de ellas en la salida estándar.

Cada línea de la entrada estándar representa una de estas figuras. Cada línea empieza con un string que indica el tipo de figura, a saber: `circulo`, `triángulo`, `rectángulo`. Dependiendo del tipo, le seguiran ciértos parámetros:

- Para `circulo`: El punto central con dos coordenadas **`int`** x e y separadas por espacios [1] y el radio (**`float`**) también separado por espacios.
- Para `triangulo`: Los tres puntos que definan el triángulo, seis enteros separados por espacios. Dos de los puntos siempre coincidirán en la coordenada y y forman parte de la base del triángulo.
- Para `rectángulo`: Los puntos sudoeste y noreste, cuatro enteros separados por espacios.

Veamos algunos ejemplos:

```
circulo 0 0 2
triangulo -1 0 2 1 0 3
rectángulo -1 -2 3 2
```

[1] Todos los puntos se representarán así.

Un poquito de ayuda:

- *Tendrás que declarar un struct por cada tipo de figura:* `circulo`*,*
- *Tendrás que utilizar union para describir que en la variable puedes tener cualquiera de las tres figuras.*
- *Necesitarás un último struct para poder discriminar el datos que tienes almacenado en el unuin*

Ejercicio 3. En la sesión de hoy, en clase, tienes múltiples ejercicios que han quedado propuestos. Te los recordamos aquí:

- Funciones sobre el tipo `enum mes`.
- Adaptar tu código a la convención de código `_t`, `_e`, `_s`, , `_u`.

Ejercicio 4. En la sesión de hoy, en clase, se ha dejado propuesto un ejercicio de ordenación de enteros en ristras. Recordatorio:

- Escribe un programa que ordene ristras de enteros de menor a mayor
- Cada ristra se representa de la siguiente forma en la entrada estánda:
  - Un entero positivo n en la primera línea
  - n enteros en las n siguientes líneas
- Por cada ristra, la salida de tu programa tiene los n enteros de la ristra ordenados de menor a mayor
- Tu programa debe parar cuando lea una ristra de 0 enteros

Veamos de nuevo un ejemplo de entrada (con dos ristras) y la salida esperada:

Entrada

```
2
7
1
3
1
4
2
0
```

Salida esperada

```

1
7

1
2
4
```

Las restricciones para este ejercicio son:

- **Usa el módulo de árboles binarios**

- **Evita consumir más memoria de la necesaria**
- **Presta especial atención a los *memory leaks***

La función de inserción en orden ya la implementaste en clase. En este ejercicio tendrás que escribir una función que haga un recorrido apropiado del árbol e imprima los enteros en orden. También tendrás que recorrer el árbol adecuadamente para liberar toda la memoria consumida.

Ejercicio 5. Entre los ejercicios de sesiones anteriores estaba la implementación del tipo abstracto de datos de las pilas: acotadas y no acotadas. Una vez que conocemos **`typedef`** y **`struct`** podemos hacerlo mucho mejor.

- En pilas acotadas podemos agrupar el array y el tamaño de la pila en un *struct*:

```
struct pila_acotada_s {
  int data[MAX];
  int top;
}

typedef struct pila_acotada_s pila_acotada_t;
```

- En pilas usando cadenas enlazadas, podemos usar las orientaciones de las transparencias:

```
/* Declaración de un struct, sólo el nombre */
struct nodo_pila_s;

/* Definición del tipo pila_t */
typedef struct nodo_pila_s *pila_t;

/* Definición del struct */
struct nodo_pila_s {
  int cima;
  pila_t resto;
};
```

Adapta las implementaciones de los ejercicios de sesiones anteriores.

Ejercicio 6. Implementar el tipo abstracto de datos de las colas (`cola_fun.c`),

- usando como estructura de datos una cadena enlazada,
- en la que el primer elemento de la cola sea el primero de la cadena,
- y respetando un interfaz *funcional* como el que indica el siguiente header:

Listing 1: cola_fun.h

```c
#ifndef COLA_FUN_H
#define COLA_FUN_H

struct nodo_cola_fun_s;

typedef struct nodo_cola_fun_s *cola_fun_t;

struct nodo_cola_fun_s {
  int primero;
  cola_fun_t resto;
}

/* Crea una nueva cola vacía */
extern cola_fun_t crear();

/* Decide si la cola está vacía */
extern int vacia(cola_fun_t cola);

/* Inserta una nuevo elemento en la cola */
extern cola_fun_t insertar(cola_fun_t cola, int dato);

/* Elimina el primero de la cola */
extern cola_fun_t borrar(cola_fun_t cola);

/* Devuelve el primero de la cosa */
extern int primero(cola_fun_t cola);

#endif
```

Ejercicio 7. Implementar el tipo abstracto de datos de las colas (cola.c),

- usando como estructura de datos una cadena enlazada,
- en la que el primer elemento de la cola sea el primero de la cadena,
- y respetando un interfaz *procedural*[2]) como el que indica el siguiente header:

Listing 2: cola.h

```c
#ifndef COLA_H
#define COLA_H

struct nodo_cola_s;

typedef struct nodo_cola_s *cola_t;
```

[2]Observa cómo cuando se quiere modificar la cola, ésta se pasa por referencia.

```

struct nodo_cola_s {
  int primero;
  cola_t resto;
}

/* Crea una nueva cola vacía */
extern void crear(cola_t *cola);

/* Decide si la cola está vacía */
extern int vacia(cola_t cola);

/* Inserta una nuevo elemento en la cola */
extern void insertar(cola_t *cola, int dato);

/* Elimina el primero de la cola */
extern void borrar(cola_t *cola);

/* Devuelve el primero de la cosa */
extern int primero(cola_t cola);

#endif
```

⏭ **Ejercicio 8.** ¿Recuerdas las órdenes bash que generaban listas de enteros? Para probar el ejercicio anterior, te sugerimos que las modifiques para generar más de una ristra de enteros.