Sesión 08: Más sobre tipos y módulos

Hoja de problemas

Programación para Sistemas

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

2022-2023

Ejercicio 1. Repasa las transparencias de clase. Además de explorar los nuevos elementos (**`enum`** y **`union`**), **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`

```
#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`

```
#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.

```
7
8  struct nodo_cola_s {
9    int primero;
10   cola_t resto;
11 }
12
13 /* Crea una nueva cola vacía */
14 extern void crear(cola_t *cola);
15
16 /* Decide si la cola está vacía */
17 extern int vacia(cola_t cola);
18
19 /* Inserta una nuevo elemento en la cola */
20 extern void insertar(cola_t *cola, int dato);
21
22 /* Elimina el primero de la cola */
23 extern void borrar(cola_t *cola);
24
25 /* Devuelve el primero de la cosa */
26 extern int primero(cola_t cola);
27
28 #endif
```

Ejercicio 8. En este ejercicio vamos a combinar conocimiento de varios temas para crear un tipo *tipo abstracto de datos*[3]. El tipo abstracto de datos que tendrás que implementar será el de las pilas acotadas de enteros.

Para implementar dicho tipo abstracto de datos tendrás que usar un array de enteros, **y nada más**. Como necesitas llevar la cuenta del número de elementos que hay en la pila y la capacidad máxima, la estructura a utilizar va a ser la siguiente:

- En la posición 0 del array guardarás la capacidad de la pila.
- En la posición 1 del array guardarás el número de elementos en la pila.
- En la posición 2 del array guardarás el primer elemento en ser apilado.
- En la posición 3 del array guardarás el segundo elemento en ser apilado.
- etc.

Vamos a construir un módulo para dicho tipo abstracto. Empezaremos con el *header*:

Listing 3: `pila_acotada.h`

```
1 #ifndef PILA_ACOTADA_H
2 #define PILA_ACOTADA_H
3
4 /* Inicializa una pila dejándola vacía */
```

[3] Aunque en C cuesta ser realmente abstracto, lo vamos a intentar.

```
5  extern void inicializar(int pila[],
6                          int capacidad);
7
8  /* Decide si la pila está vacía */
9  extern int vacia(int pila[]);
10
11 /* Decide si la pila está vacía */
12 extern int llena(int pila[]);
13
14 /* Coloca x en la cima de la pila (si la pila no está llena */
15 extern void apilar(int pila[], int x);
16
17 /* Devuelve la cima de la pila (si la pila no está  vacía) */
18 extern int cima(int pila[]);
19
20 /* Elimina el elemento en la cima de la pila */
21 extern void desapilar(int pila[]);
22
23 #endif
```

Tu labor será completar e implementar correctamente las funciones en el fichero `pila_acotada.c`. Te ofrecemos aquí el esqueleto:

Listing 4: `pila_acotada.c`

```c
#include "pila_acotada.h"

void inicializar(int pila[],
                 int capacidad){
  /* Implementar correctamente */
}

int vacia(int pila[]) {
  /* Implementar correctamente */
  return 0;
}

int llena(int pila[]) {
  /* Implementar correctamente */
  return 0;
}

void apilar(int pila[], int x) {
  /* Implementar correctamente */
}

int cima(int pila[]) {
  /* Implementar correctamente */
  return 0;
}

void desapilar(int pila[]) {
  /* Implementar correctamente */
}
```

Para ayudarte a saber que vas por el buen camino hemos elaborado el siguiente programar de test:

Listing 5: `pila_acotada_test.c`

```c
#include <assert.h>
#include <stdio.h>
#include "pila_acotada.h"

#define MAX 1000
#define N 20

int main() {
  int pila[MAX];
  int i;

  /* Para no cagarla con los parámetros de la prueba */
  assert(N < MAX);
  assert(N > 1);

  inicializar(pila, N);

  /* Tras inicializar la pila debería estár vacía */
  assert(vacia(pila));
  /* y por tanto no llena */
  assert(!llena(pila));

  apilar(pila,42);

  /* Tras apilar un elemento la pila no debería estar vacía */
  assert(!vacia(pila));
  /* tampoco llena (N es mayor que 1) */
  assert(!llena(pila));
  /* y la cima debería ser el 42 apilado */
  assert(cima(pila) == 42);

  desapilar(pila);

  /* Tras desapilar el único elemento la pila debería estar vacía */
  assert(vacia(pila));
  /* y no llena */
  assert(!llena(pila));

  /* Este bucle casi llena la pila */
  for (i = 1; i < N; i++) {
    apilar(pila, 2*i);
    /* Tras cada apilado la pila no debería estar vacía */
    assert(!vacia(pila));
```

```
    /* tampoco llena */
    assert(!llena(pila));
    /* y la cima debería ser lo último apilado */
    assert(cima(pila) == 2*i);
  }

  apilar(pila, 0);

  /* Tras apilar un elemento más, la pila no debería estar vacía */
  assert(!vacia(pila));
  /* y ahora sí, ya debería estar llena */
  assert(llena(pila));
  /* Y la cima debería ser lo último apilado */
  assert(cima(pila) == 0);

  desapilar(pila);

  /* Este bucle casi vacía la pila */
  for (i = N - 1; i > 1; i--) {
    /* Tras cada desapilado la pila debería tener los elementos
       apilados en orden inverso */
    assert(cima(pila) == 2*i);

    desapilar(pila);

    /* Tras cada desapilado la pila no deberia estar vacía */
    assert(!vacia(pila));
    /* pero tampoco llena */
    assert(!llena(pila));
  }

  desapilar(pila);

  /* Tras el último desapilado, la pila deberia estar vacía */
  assert(vacia(pila));
  /* y no llena */
  assert(!llena(pila));

  fprintf(stderr, "¡Todos los tests pasados!\n");

  return 0;
}
```

Y para que lo tengas aún más fácil, puedes utilizar el siguiente `Makefile`:

```
CFLAGS=-Wall -Werror -g -pedantic

pila_acotada_test: pila_acotada.o pila_acotada_test.o
	$(CC) $(CFLAGS) -o $@ $^

pila_acotada.o: pila_acotada.c pila_acotada.h

test: pila_acotada_test
	./pila_acotada_test

clean:
	rm -f *.o pila_acotada_test
```

Cuando todo esté correcto deberías experimentar la siguiente sesión Bash:

```
$ make
cc -Wall -Werror -g -pedantic   -c -o pila_acotada.o pila_acotada.c
cc -Wall -Werror -g -pedantic   -c -o pila_acotada_test.o pila_acotada_test.c
cc -Wall -Werror -g -pedantic -o pila_acotada_test pila_acotada.o pila_acotada_test.o
$ ./pila_acotada_test
¡Todos los tests pasados!
$ |
```

Ejercicio 9. Modifica tu implementación de *polaca inversa* `rpn` para usar tus recién estrenadas pilas acotadas.

Ejercicio 10. ¿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.

Ejercicio 11. Tu labor en este ejercicio será modificar el módulo `generador_lcg` y enriquecerlo de la siguiente forma.

- La idea es tener un módulo desde el que poder cambiar en tiempo de ejecución los parámetros a, c y m del generador (ahora no es posible porque se han definido como macros).
- El módulo va a contener tres variables: `lcg_a`, `lcg_c`, y `lcg_m`.
- Además, se expondrá una operación para establecer la *semilla* del generador, es decir, el valor de X_0.

Para facilitar la tarea dispones del *header* aquí:

generador_lcg.h

```
/* Parámetros a, c, y m del generador */
extern int lcg_a = 1;
```

```
extern int lcg_c = 1;
extern int lcg_m = 1;

/* Resetea el generador colocando el valor de X_0 al valor de semilla */
extern void lcg_resetear(int semilla);

/* Devuelve el valor de X y genera el siguiente */
extern int lcg_generar();
```

Para que puedas ver si lo que has hecho funciona correctamente puedes usar este *tester*:

test_lcg.h

```
#include <stdio.h>
#include <assert.h>
#include "generador_lcg.h"
int main() {

  lcg_a = 7;
  lcg_c = 1;
  lcg_m = 11;

  lcg_resetear(9);

  assert(lcg_generar() == 9);
  /* 7 * 9 + 1 % 11 == 9 */
  assert(lcg_generar() == 9);

  lcg_resetear(0);

  assert(lcg_generar() == 0);
  /* 7 * 0 + 1 % 11 == 1 */
  assert(lcg_generar() == 1);
  /* 7 * 1 + 1 % 11 == 8 */
  assert(lcg_generar() == 8);
  /* 7 * 8 + 1 % 11 == 2 */
  assert(lcg_generar() == 2);
}
```