FAQ - Lenguaje C

¿Para qué se usa el tipo size_t?

`size_t` es un tipo entero sin signo devuelto por el operador `sizeof` y es usado para representar el tamaño de construcciones en bytes. Este tipo está definido de manera tal de garantizar que siempre va a poder almacenar el tamaño del tipo más grande posible, por lo que también garantiza que va a poder almacenar cualquier índice de cualquier arreglo.

Estas características lo convierten en el tipo adecuado para manejar tamaños e índices.

Cadenas

¿Cómo comparo si dos cadenas son iguales?

En C las cadenas de caracteres son vectores que tienen caracteres como elementos. C no sabe comparar vectores, de ningún tipo. Al hacer una comparación del tipo: `cadena1 == cadena2`, lo que se compara es que las direcciones de memoria de ambas variables sean iguales, es decir que sólo van a ser iguales cuando realmente sean el mismo puntero.

Para poder comparar el contenido de dos cadenas, es necesario usar la función `strcmp(cadena1, cadena2)`, que devuelve 0 si son iguales, menor que 0 si la primera es menor y mayor que 0 si la primera es mayor. En este caso no importa que las cadenas ocupen o no la misma porción de memoria.

¿Cómo copio dos cadenas?

La sentencia:

```
char* cad_1 = cad_2;
```

No crea una copia de una cadena, sino una copia de la referencia a la cadena. Para hacer una copia de una cadena es necesario hacer:

```
strcpy(buf_destino, cad_origen);
```

Siendo `buf_destino` una posición de memoria tal que pueda albergar la cadena a copiar, típicamente reservada con `malloc` y `strlen` (teniendo en cuenta el espacio necesario para alojar el fin de cadena).

Desde el curso se recomienda la creación de una función auxiliar `strdup` que encapsule esta lógica.

Pueden leer más sobre sobre cadenas en el apunte del curso.

¿Qué es getline? ¿Cómo uso getline?

`getline()` es una función que lee una linea de un archivo.

```
ssize_t getline(char** buffer, size_t* capacidad, FILE* archivo);
```

La principal ventaja de esta función es que automáticamente reserva la memoria dinámica necesaria para almacenar la línea. No se necesita invocar manualmente malloc() a mano, ni preocuparse por los tamaños de los buffers.

Sí se necesita, no obstante, liberar memoria al terminar: getline() llama a malloc() pero transfiere la responsabilidad de la memoria al usuario.

La reserva de memoria dinámica se consigue cuando el puntero buffer apunta a NULL, y el puntero capacidad apunta a 0:

```
char* buffer = NULL; size_t capacidad = 0;
ssize_t leidos = getline(&buffer, &capacidad, archivo);
// Hago uso de la linea
free(buffer);
```

La función getline() se encuentra definida en la cabecera stdio.h. Como es una función de POSIX.1-2008, hay que declarar un identificador para indicar que la queremos usar:

```
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
```

Para más información y especificaciones de la función, se puede invocar al comando `man` en la terminal, para ver el manual de esta. `man getline`.

¿Qué es POSIX?

Portable Operating System Interface (POSIX) es una familia de estándares especificados por el Instituto de Ingeniería Eléctrica y Electrónica (IEEE), una asociación sin fines de lucro dedicada a estandarización, para mantener la compatibilidad entre distintos sistemas operativos.

Al hacer `#include <stdio.h>` lo que le decimos al pre-procesador es que en mi programa yo quiero poder utilizar lo definido por C en esa cabecera de la biblioteca estándar.

Por fuera de la existencia del estándar de C que define y especifica las funciones que utilizamos en la biblioteca estándar (`stdio.h`, `stdlib.h`, etc.), el estándar POSIX lo extiende con funciones y comportamientos nuevos.

Por otro lado, al hacer `#define _POSIX_C_SOURCE 200809L` y luego `#include <stdio.h>` (respetar el orden), lo que le decimos al pre-procesador es que en mi programa yo quiero poder utilizar la especificación de POSIX de 2008 de la biblioteca estándar (de C). Entre otras cosas, esta especificación incluye `getline()` y `getdelim()`.

¿Qué es un puntero a función?

Los punteros a función son variables que apuntan a funciones para que puedan ser invocadas sin conocer su nombre. En C, la sintaxis para declararlas es diferente al resto de los tipos. Por ejemplo, dada la siguiente función, que recibe dos cadenas y devuelve un entero:

```
int mi_funcion(char *a, char *b) {
    return a[0] == b[0];
}
```

Es posible declarar una variable cuyo tipo sea "puntero a función que recibe dos cadenas y devuelve un entero", y asignar a esa variable la dirección de la función:

```
int (*mi_puntero)(char*, char*);  // Declara la variable (puntero).
mi_puntero = mi_funcion;          // Asigna un valor al puntero.
```

Una vez se ha inicializado la variable correctamente, se puede invocar como si el puntero mismo fuera una función:

```
int ret = mi_puntero("hola", "adios");
```

Estas funciones se usan típicamente para delegar la ejecución de un fragmento de código a otra función. Por ejemplo, para aplicarle una operación a todos los elementos de un arreglo de enteros podríamos tener la función:

```
/* Aplica la operación pasada por parámetro a todos los elementos del arreglo. */
void aplicar(int arreglo[], size_t cant, int (*operacion)(int));
```

Entonces si tuviéramos una función para multiplicar por dos:

```
int multiplicar_por_dos(int numero)
{
    return numero * 2;
}
```

Podríamos multiplicar por dos todos los elementos del arreglo usando un puntero a la función:

```
int arreglo[3] = { 2, 5, 4 };
aplicar(arreglo, 3, multiplicar_por_dos);
```

¿Qué es un wrapper?

En la materia usamos los punteros a función para garantizar un comportamiento que no dependa del tipo de datos que se maneje. Por ejemplo, si quisiéramos destruir todos los elementos de una estructura podríamos crear una funcion:

```c
void estructura_destruir(estructura_t* estructura, void (*f_dest)(void*));
```

que le aplique la función de destrucción a todos los elementos que están almacenados en ella. Si estos elementos también son genéricos y tienen su propia función de destrucción:

```c
void elemento_destruir(elemento_t* elem);
```

No se puede invocar directamente a la primitiva `estructura_destruir` con `elemento_destruir`, porque sus firmas son diferentes. Lo que se suele hacer es crear una función wrapper que enmascare el comportamiento:

```c
void elemento_destruir_wrapper(void* elem) {
    elemento_destruir(elem);
}
```

De esta manera, la función `elemento_destruir_wrapper` es genérica y puede ser usada con la función `estructura_destruir`.

¿Qué es typedef?

Typedef es una característica de C que nos permite darle un alias a cualquier tipo de C. Por ejemplo, si quisiéramos representar la edad de una persona podríamos querer abstraernos de si representamos esa edad como un entero, un entero sin signo, un entero corto (`short`), etc. y además tener consistencia en todo nuestro código de usar siempre el mismo tipo para todas las variables que representen una edad. Para esto podemos 'crear' un tipo nuevo en C que se refiera a la edad de las personas, supongamos que como base quisiéramos hacer uso del tipo `unsigned int` para crear nuestro tipo `edad`, pero a su vez queremos dejarle explícito al lector que éste es un tipo definido por el programador, por lo que por convención le agregamos el sufijo `_t` (que se lee como "tipo") siendo nuestro nuevo tipo `edad_t`.

Para esto la sintaxis correspondiente es:

```c
typedef unsigned int edad_t;
```

Siempre la sintaxis para definir un nuevo tipo en C *es como* si se tratara de declarar una variable con determinado nombre, pero anteponiendo la palabra reservada `typedef`. Entonces, en vez de declarar una nueva variable lo que se "declara" es un nuevo tipo, es decir: `typedef tipo_existente mi_nuevo_tipo;`.

Y luego podemos, como si fuese cualquier otro tipo de datos de C, declarar variables, hacer uso de ellas y demás como:

```c
edad_t luis = 14;
// Equivale a unsigned int luis = 14
```

La **ventaja** de utilizar `typedef` es que se agrega una capa de abstracción sobre el tipo: en vez de preguntarme qué representa determinada variable de tipo `unsigned int`, el mismo tipo `edad_t` me aclara exactamente para qué sirve ese tipo. *Sintácticamente* escribir `unsigned int` y `edad_t` en mi código van a ser exactamente lo mismo, pero *semánticamente* son dos tipos totalmente diferentes y no son intercambiables. Además de la abstracción simplifica la mantenibilidad del código, por ejemplo, si el día de mañana hiciera la asunción de que la edad de las cosas no puede superar los 255 años, tranquilamente podría reemplazar la definición del `typedef` por un `unsigned char` y toda mi implementación se actualizaría consistentemente a esa nueva representación apenas modificando una línea en mi código.

Por otro lado la **desventaja** de utilizar `typedef` es que se enmascaran los tipos básicos. Entonces cuando vemos una variable de tipo `edad_t` tal vez en vez de abstraernos tenemos que ir a buscar su definición para saber si es un tipo entero, una estructura, etc. y cómo definirla o utilizarla. A veces *menos es más*.

`typedef` puede servirnos para otras cosas, como para omitir la palabra `struct` cada vez que hacemos una estructura en C, dándole ahora a nuestras estructuras el rango de tipo:

```c
struct persona {
    edad_t edad;
    char *nombre;
};

typedef struct persona persona_t;
```

Cuando se utilizan estructuras para encapsular tipos abstractos de datos suele mantenerse de forma privada la definición de la estructura. En estos casos la definición de la estructura queda en el archivo *.c* mientras que la definición del tipo se publica en una cabecera *.h*, permitiendo que la interfaz pública del dato use el nombre del tipo.

Si se trata de un `struct` que no se expone públicamente (por ejemplo, por estar limitado a un único *.c* y declararse ahí), se puede combinar declaración y `typedef` en una sola instrucción:

```
typedef struct {
    edad_t edad;
    char *nombre;
} persona_t;
```

Como ejemplo también podemos hacer uso de `typedef` para simplificar la declaración de punteros a función:

```
// recibe_puntero_a_funcion es una funcion que recibe un puntero
// funcion_recibida es un puntero a funcion que recibe void* y devuelve bool

// Versión A, sin typedef.
int recibe_puntero_a_funcion(bool (*funcion_recibida)(void*));

// Version B, con typedef
// Hago un tipo de funciones, funcionvoidp_t que se refiere a las funciones que
// reciben void* y devuelven bool

typedef bool (*funcionvoidp_t)(void*);

int recibe_puntero_a_funcion(funcionvoidp_t funcion_recibida);
```