Guía de Serialización

Objetivo

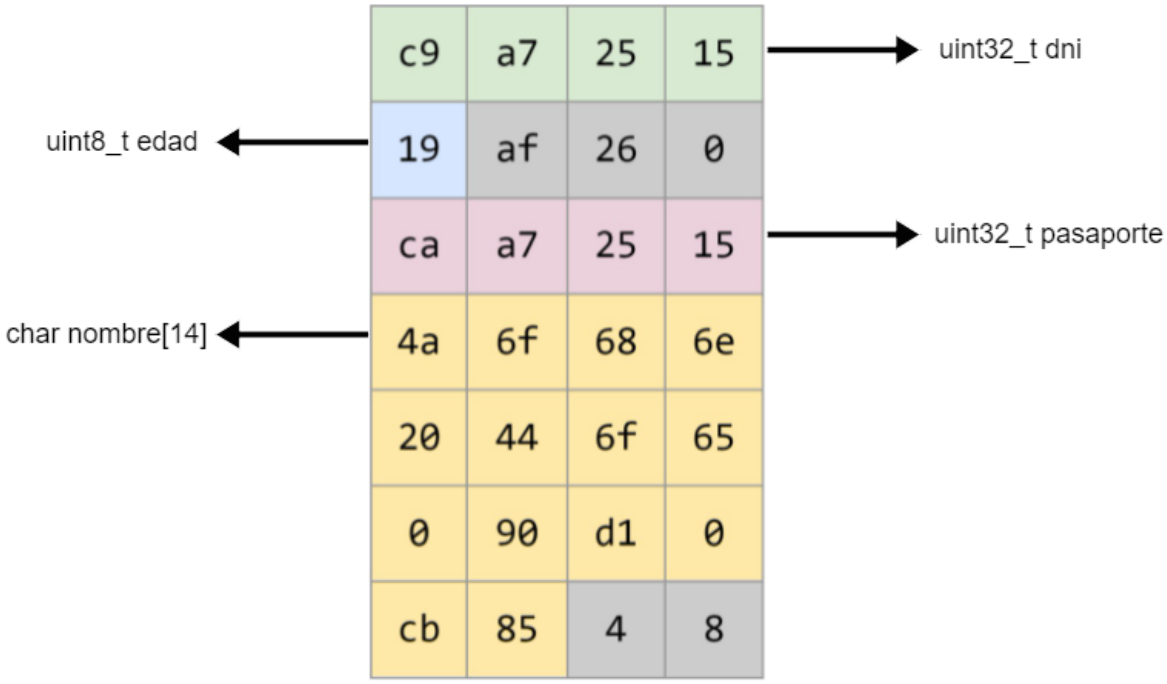
El presente documento tiene como fin explicar qué es, para qué sirve y cómo serializar distintos tipos de mensaje desde C. Para esto, se explicarán los distintos tipos de mensajes y se ofrecerán ejemplos de cómo serializarlos para enviarlos.

TADs

Decimos que un TAD (tipo abstracto de dato) es un conjunto de datos que tiene operaciones asociadas, que se encuentran escondidos bajo un mismo identificador. Generalmente en C, representamos un TAD mediante el uso de un `struct` .

A la hora de trabajar con structs podemos notar que el tamaño de la misma en memoria no siempre coincide con la suma del tamaño de sus elementos (es más, no suele ser así). Esto ocurre a causa del **padding** que agrega el compilador para intentar optimizar los accesos a memoria.

```c
typedef struct {
    uint32_t dni;
    uint8_t edad;
    uint32_t pasaporte;
    char nombre[14];
} t_persona;

t_persona p1;
p1.dni=35478921;
p1.edad=25;
p1.pasaporte=3547899322;
strcpy(p1.nombre,"John Doe");
```



El tamaño de este padding depende de varios factores como, por ejemplo, el compilador, la arquitectura del CPU, el tipo de dato, entre otros. Por esto, es que no tenemos garantías de cuánto espacio ocupará un struct en memoria.

¿En qué nos afecta esto a nosotros? Cuando querramos mandar mensajes a través de la red, nunca sabremos cuál es el tamaño real de nuestra estructura; porque el padding, como dijimos anteriormente, depende de muchos factores. Si vemos el prototipo de las funciones `send()` y `recv()`, podemos notar que ambas requieren que les digamos cuántos bytes vamos a enviar o recibir y si nosotros, usamos `sizeof(miTAD)` no tenemos garantías de que el tamaño de mi struct coincida entre el emisor y el destinatario. Es decir, si tomamos el ejemplo del struct que se encuentra arriba, puede que el `sizeof(t_persona)` en nuestra computadora dé X número pero, cuando lo enviemos y lo reciba otra computadora, le dé otro número provocando que exista la posibilidad de que se pierdan datos.

Para solucionar estos inconvenientes debemos **serializar**.

¿Qué es serialización?

El concepto de serializar consiste en *especificar un protocolo en común, para que dos procesos se comuniquen de manera organizada*. La idea es poder enviar un stream de datos siguiendo un orden definido y conocido por ambos procesos.

Estructuras estáticas

Supongamos que tenemos el siguiente TAD:

```c
typedef struct {
    uint32_t dni;
    uint8_t edad;
    uint32_t pasaporte;
    char nombre[14];
```

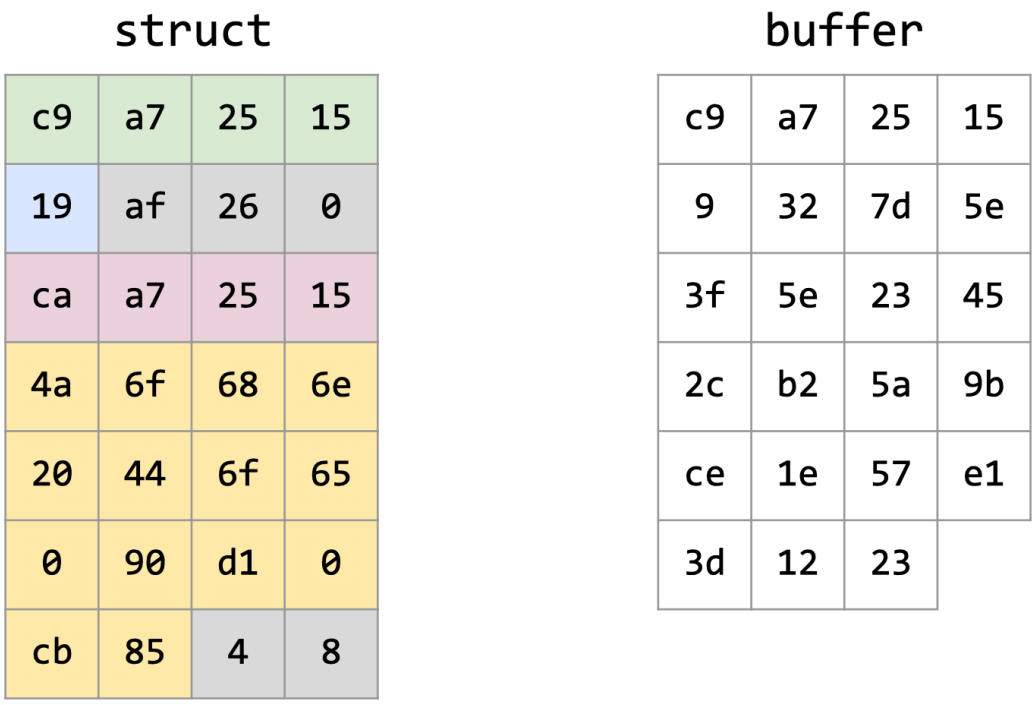
```
7 } t_persona;
8
```

Nuestro objetivo entonces será organizar los elementos para enviarlos de una manera ordenada utilizando un protocolo:

En este caso, un posible protocolo, es agregar un header que normalmente utilizaremos para indicar qué tipo de dato vamos a enviar. La idea es que del otro lado se reciba primero este header, y con eso ya se puede "preparar" para saber qué viene después (generalmente todo lo que viene después del header se conoce como *payload*).

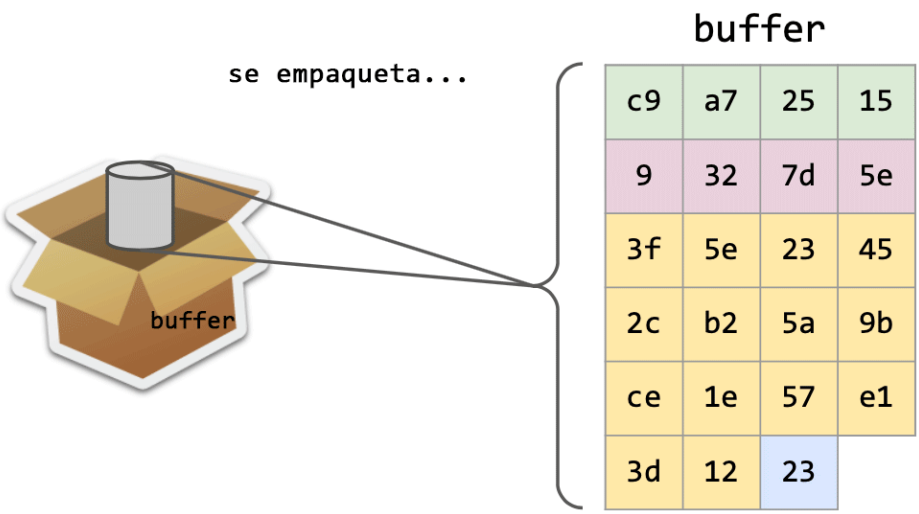
Es decir, si en el header indicamos que vamos a enviar una "persona", del otro lado al recibir esto ya sabe que lo próximo a recibir va a ser el DNI, el pasaporte, el nombre y, por último, la edad.

Entonces, ¿cómo hacemos para serializar este struct? Primero empezamos reservando un bloque de HEAP, suficiente para almacenar nuestra estructura. Éste lo denominaremos como nuestro buffer intermedio, donde se irán guardando nuestros datos mediante la utilización de la función `memcpy()`, es decir, se asignará todo lo que se encuentra en la estructura, en ese buffer.



De aquella manera, se ignorará el padding que el compilador crea, y se guardarán solamente los datos que necesitamos de nuestra estructura.

Una vez hecho esto, se podrá empaquetar el buffer y se enviará a través de la red, garantizando que los datos que recibirá el destinatario serán los correctos.



Una vez que el destinatario los reciba, los deberá desempaquetar de forma inversa (esto es posible dado que se determinó un protocolo).

Estructuras dinámicas

Hasta ahora logramos serializar una estructura estática, es decir, una estructura de la cual conocemos el tamaño de todos los elementos de nuestro TAD. *¿Qué pasa si tenemos una estructura dinámica, donde no sabemos el tamaño del contenido que vamos a enviar?* Por ejemplo, una estructura que tenga un dato, o varios, que sea, por ejemplo, un texto cuyo tamaño no conocemos hasta que el programa esté corriendo.

¿Hacemos lo mismo que hicimos con estructuras estáticas? No. *¿Por qué?* En principio no se sabe cuántos bytes tiene que recibir del otro lado para formar el paquete, es decir, teniendo en cuenta la siguiente estructura:

```c
typedef struct t_Package {
    char* username;
    char* message;
} t_package;
```

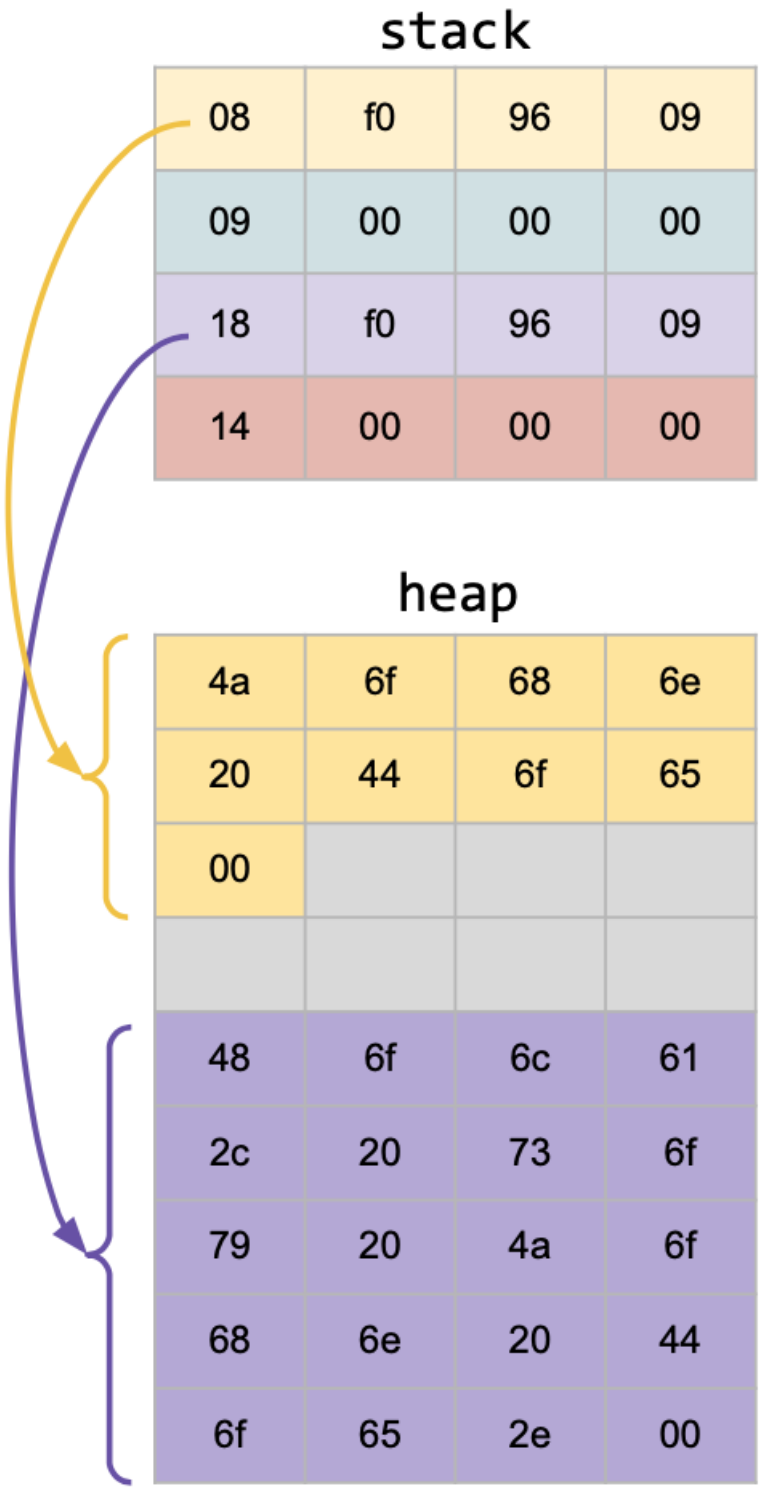
No tenemos un `char[14]`, tenemos un puntero `(char*)` que apunta a una posición de memoria provocando que haya estructuras que van a tener usernames de 14 bytes, otras de 20 bytes, y así, porque vamos a reservar memoria dinámicamente según la longitud del username que se quiera.

Si seguimos los pasos anteriores, el receptor de nuestra estructura no va a saber cuánta memoria deberá reservar para poder leer lo que recibió. Es decir, si nosotros enviamos un username de 14 bytes, lo empaquetamos y lo enviamos, el receptor recibirá el paquete, pero a la hora de desempaquetarlo, que debemos realizar un `memcpy()`, no sabrá cuánta memoria deberá reservar para poder leer lo que le enviamos.

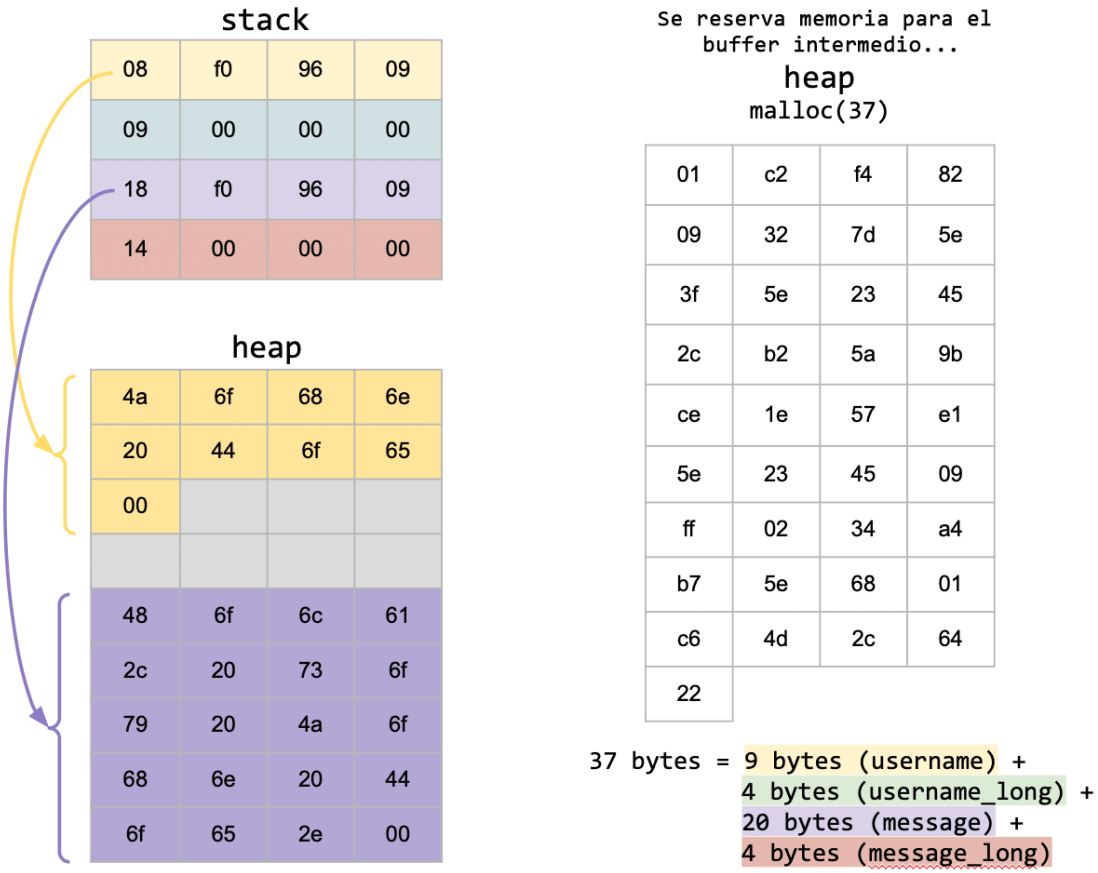
¿Cómo lo solucionamos? Nuestra recomendación es colocarle a nuestro protocolo el tamaño en bytes que necesitará el destinatario para almacenar lo que enviamos, es decir, teniendo la siguiente estructura, establecer el siguiente protocolo:

c

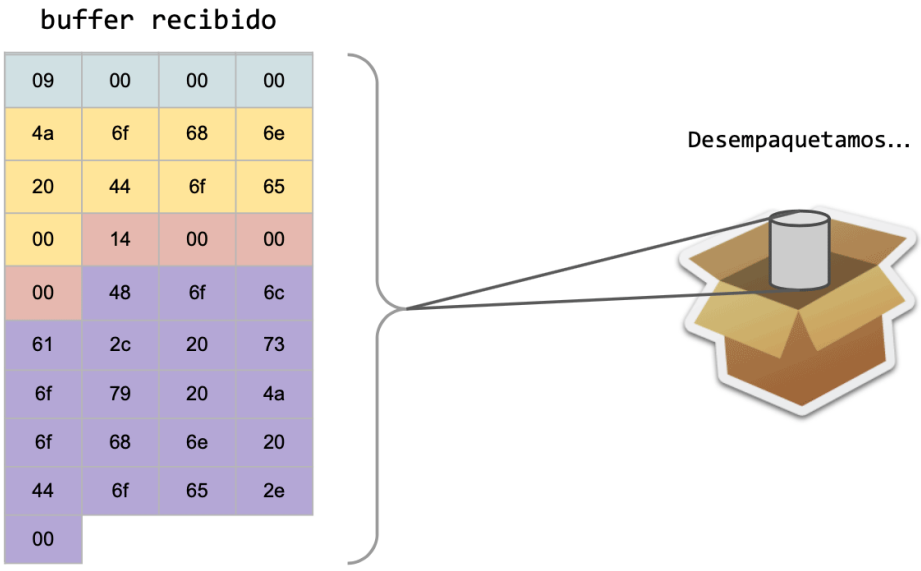
```c
typedef struct t_Package {
    char* username;
    uint32_t username_long;
    char* message;
    uint32_t message_long;
} t_package;

t_package p1;
p1.username = malloc(8+1);
p1.message = malloc(19+1);

strcpy(p1.username, "John Doe");
p1.username_long = strlen(p1.username)+1;

strcpy(p1.message, "Hola, soy John Doe.");
p1.message_long = strlen(p1.message)+1;
```



De aquella forma, luego se proseguirá con los mismos pasos que con una estructura estática: hacemos `memcpy()` de los datos de nuestra estructura a un buffer intermedio, lo empaquetamos y lo enviamos.



El receptor cuando desempaqueta el paquete, se fijará cuántos bytes debe reservar dinámicamente, y luego proseguirá a realizar el `memcpy()` .



Hay que tener en cuenta que el principal problema de las estructuras dinámicas no son los punteros, dado que no se está enviando una posición de memoria, sino que se está enviando un valor en bytes.

Puede suceder que la computadora con la cual estamos enviando nuestros mensajes, tenga un sistema operativo de 32 bits, pero la computadora receptora de nuestros mensajes enviados, tenga un sistema operativo de 64 bits. ¿A qué vamos con esto? A que no nos importa si el tamaño de nuestro puntero coincide con quién va a recibir el mensaje, ni tampoco la posición de memoria a la que apunta inicialmente, sino que nos interesa el valor que contiene esa posición de memoria, y enviarlo de forma tal que, sin importar estas diferencias, se reciba lo enviado de forma correcta.

En C

Repaso

Antes de ver cómo lo llevamos a código, hagamos un repaso de C. Comencemos con los punteros. Se define a un puntero como a *una variable que, a diferencia de otras, contiene una dirección de memoria.* Usualmente utilizamos los punteros para, no solo apuntar a ciertas variables, sino también para el manejo de memoria dinámica.

Llevándolo un poco más a código, primero veamos cómo se utilizan los punteros para que apunten a ciertas variables:

```c
int un_nro; // Variable entera
int *int_ptr; // Puntero a entero
un_nro = 2;
```

¿Cómo hacemos para que el puntero apunte a la variable entera? Utilizando el operador `&` (ampersand). *¿Por qué no podemos simplemente igualarlo a la variable* `un_nro` *?* Porque, como dijimos antes, un puntero apunta a una dirección de memoria, por ende, si nosotros lo igualamos a una variable, no coinciden los tipos. El operador `&` indica la dirección de un objeto, es decir, desde un entero hasta un struct (TAD).

A su vez, también existe el operador `*(asterisco)` , el cual se podrá utilizar solamente en punteros, y nos permite visualizar lo que apunta, es decir, el número en este caso.

```c
int_ptr = &un_nro;
printf("int_ptr=%p\n",int_ptr); // Por pantalla se visualizará la posición de memoria de la variab
printf("*int_ptr=%d\n",*int_ptr); // Por pantalla se visualizará "lo que tiene dentro" nuestro pun
printf("&int_ptr=%p\n",&int_ptr); // Por pantalla se visualizará la dirección de memoria de nuestr
```

Hay que tener en cuenta que si nosotros modificamos algo desde nuestro puntero, es decir, en este ejemplo decidimos incrementar la variable desde el puntero, dicho cambio se verá también en nuestra variable `un_nro` .

```c
(*p)++; // n = 3
```

Respecto a la memoria dinámica, nosotros mediante la utilización de `malloc()` (en C++ se utiliza la función `new(..)`), podemos reservar memoria dinámicamente. La firma de `malloc` es la siguiente:

```c
void *malloc(size_t size);
```

Por ende, se debe especificar la cantidad de bytes que vamos a reservar utilizando el operador `sizeof` .

Utilizando la estructura que utilizamos anteriormente, vamos a ver cómo se manejan estas estructuras con los punteros.

```c
typedef struct {
    uint32_t dni;
    uint8_t edad;
    uint32_t pasaporte;
    char nombre[14];
} t_persona;

t_persona *ptr_persona = (t_persona*) malloc(sizeof(t_persona));
// Se reserva memoria para poder apuntar a una estructura
```

Hay dos formas de acceder a la estructura a través del puntero, las cuales son las siguientes y son iguales, no hay ninguna diferencia entre ambas y cumplen la misma finalidad:

```c
(*ptr_persona).dni = 41543667;
ptr_persona->dni = 41543667;
```

Y, para concluir con el repaso, hay que tener en cuenta que cada vez que se reserve memoria, habrá que liberarla. En caso de no liberarse se dice que se produce un *memory leak*.

Esto es muy importante dado que se tiene en cuenta a la hora de la evaluación del trabajo práctico la cantidad de memoria perdida.

¿Cómo se libera memoria en C? Se utiliza la función `free()` cuya firma es:

```c
void free(void *ptr);
```

Por ende, para liberarlo solamente se deberá pasar por parámetro el puntero. Sin embargo, hay que tener en cuenta que, si nuestra estructura es una estructura dinámica, se deberá liberar previamente la memoria reservada por los punteros de la estructura, dado que si se se libera el puntero a la estructura, es decir `t_package*` tomando el ejemplo anterior, la memoria reservada por los punteros internos nunca se liberaran, y se producira un memory leak.

```c
free(ptr_persona);
// Si se tuviese la estructura dinámica ejemplificada anteriormente, se deberá hacer lo mismo con
free((*ptr_paquete).username);
free((*ptr_paquete).message);
free(ptr_paquete);
```

Serialicemos

Finalmente, traslademos la serialización a la práctica, es decir, veamos cómo trabajar con todo esto en el código.

Para ello utilizaremos la siguiente estructura:

```c
typedef struct {
    uint32_t dni;
    uint8_t edad;
    uint32_t pasaporte;
    uint32_t nombre_length;
    char* nombre;
} t_persona;
```

Lo que haremos es usar un buffer temporal donde armar nuestro paquete. En este caso tendremos:

- 4 bytes del dni;
- 1 byte de la edad;
- 4 bytes del pasaporte;
- N bytes del nombre;
- 4 bytes de la longitud (N) del nombre.

El buffer podría ser un simple `void*` reservado con `malloc()`. Sin embargo, para esta guía, vamos a armar un `struct t_buffer` que, además del puntero a nuestro stream de datos, tendrá el tamaño en bytes del mismo stream. Esto nos permitirá poder recibir el payload entero desde el lado receptor en una sola operación, puesto que conoceremos su tamaño.

```c
typedef struct {
    uint32_t size; // Tamaño del payload
    void* stream; // Payload
} t_buffer;
```

Para ir llenando este buffer, como vimos anteriormente, utilizaremos la función `memcpy()`, que nos permite copiar bytes desde un bloque de memoria origen a uno de destino:

```c
void *memcpy(void *dest, const void *src, size_t n);
```

Ahora, si tenemos una variable declarada `(t_persona persona)` con los datos de a enviar en ella, entonces llenar nuestro buffer podría ser algo como lo siguiente:

```c
t_buffer* buffer = malloc(sizeof(t_buffer));

buffer->size = sizeof(uint32_t) * 3 // DNI, Pasaporte y longitud del nombre
             + sizeof(uint8_t) // Edad
             + strlen(persona.nombre) + 1; // La longitud del string nombre. Le sumamos 1 para env

void* stream = malloc(buffer->size);
int offset = 0; // Desplazamiento

memcpy(stream + offset, &persona.dni, sizeof(uint32_t));
offset += sizeof(uint32_t);
memcpy(stream + offset, &persona.edad, sizeof(uint8_t));
offset += sizeof(uint8_t);
memcpy(stream + offset, &persona.pasaporte, sizeof(uint32_t));
offset += sizeof(uint32_t);
```

```c

// Para el nombre primero mandamos el tamaño y luego el texto en sí:
memcpy(stream + offset, &persona.nombre_length, sizeof(uint32_t));
offset += sizeof(uint32_t);
memcpy(stream + offset, persona.nombre, strlen(persona.nombre) + 1);
// No tiene sentido seguir calculando el desplazamiento, ya ocupamos el buffer completo

buffer->stream = stream;

// Si usamos memoria dinámica para el nombre, y no la precisamos más, ya podemos liberarla:
free(persona.nombre);
```

Para ir moviéndonos en el buffer utilizamos una variable de desplazamiento (*offset*). La idea es que si tenemos un puntero, podemos "sumarle/restarle" valores y eso nos da otro puntero. Esto se conoce comúnmente como aritmética de punteros. Sumándole un valor n a un puntero de un tipo de dato que ocupa *m* bytes se obtiene un puntero a una dirección desplazada *n * m* bytes respecto de la original.

Si por ejemplo tenemos un `int* a` y hacemos `a + 1` nos estamos desplazando *4 + 1 = 5* bytes respecto de la dirección apuntada por `a`. En este caso utilizamos el tipo `void*` para nuestro buffer, ya que por lo general se interpreta que los elementos a los que apunta ocupan 1 byte. De esta manera haciendo `buffer + offset` nos desplazamos tantos bytes respecto del inicio del buffer como indique nuestra variable `offset`.

Bien, ahora en el buffer ya tenemos nuestro TAD persona cargado, podríamos ya enviar esto. Pero estaría bueno "empaquetarlo" primero, es decir, agregarle un header al principio. De esta manera el receptor viendo el header ya sabe que le están mandando una persona y se puede preparar para recibir el resto de datos. Así podemos enviar otros TAD distintos, y el receptor sabrá como deserializarlo al revisar el header.

```c
typedef struct {
    uint8_t codigo_operacion;
    t_buffer* buffer;
} t_paquete;
```

Entonces podemos llenar el paquete con el buffer:

```c
t_paquete* paquete = malloc(sizeof(t_paquete));

paquete->codigo_operacion = PERSONA; // Podemos usar una constante por operación
paquete->buffer = buffer; // Nuestro buffer de antes.

// Armamos el stream a enviar
void* a_enviar = malloc(buffer->size + sizeof(uint8_t) + sizeof(uint32_t));
int offset = 0;

memcpy(a_enviar + offset, &(paquete->codigo_operacion), sizeof(uint8_t));

offset += sizeof(uint8_t);
memcpy(a_enviar + offset, &(paquete->buffer->size), sizeof(uint32_t));
offset += sizeof(uint32_t);
memcpy(a_enviar + offset, paquete->buffer->stream, paquete->buffer->size);

// Por último enviamos
```

```c
send(unSocket, a_enviar, buffer->size + sizeof(uint8_t) + sizeof(uint32_t), 0);

// No nos olvidamos de liberar la memoria que ya no usaremos
free(a_enviar);
free(paquete->buffer->stream);
free(paquete->buffer);
free(paquete);
```

Bien, ya enviamos nuestro mensaje. Ahora, ¿cómo lo deserializamos desde el lado del receptor?

```c
t_paquete* paquete = malloc(sizeof(t_paquete));
paquete->buffer = malloc(sizeof(t_buffer));

// Primero recibimos el codigo de operacion
recv(unSocket, &(paquete->codigo_operacion), sizeof(uint8_t), 0);

// Después ya podemos recibir el buffer. Primero su tamaño seguido del contenido
recv(unSocket, &(paquete->buffer->size), sizeof(uint32_t), 0);
paquete->buffer->stream = malloc(paquete->buffer->size);
recv(unSocket, paquete->buffer->stream, paquete->buffer->size, 0);

// Ahora en función del código recibido procedemos a deserializar el resto
switch(paquete->codigo_operacion) {
    case PERSONA:
        t_persona* persona = deserializar_persona(paquete->buffer);
        ...
        // Hacemos lo que necesitemos con esta info
        // Y eventualmente liberamos memoria
        free(persona);
        ...
        break;
    ... // Evaluamos los demás casos según corresponda
}

// Liberamos memoria
free(paquete->buffer->stream);
free(paquete->buffer);
free(paquete);
```

Ahora nuestra función para deserializar el payload quedaria algo asi:

```c
t_persona* deserializar_persona(t_buffer* buffer) {
    t_persona* persona = malloc(sizeof(t_persona));

    void* stream = buffer->stream;
    // Deserializamos los campos que tenemos en el buffer
    memcpy(&(persona->dni), stream, sizeof(uint32_t));
    stream += sizeof(uint32_t);
    memcpy(&(persona->edad), stream, sizeof(uint8_t));
    stream += sizeof(uint8_t);
    memcpy(&(persona->pasaporte), stream, sizeof(uint32_t));
    stream += sizeof(uint32_t);

    // Por último, para obtener el nombre, primero recibimos el tamaño y luego el texto en sí:
    memcpy(&(persona->nombre_length), stream, sizeof(uint32_t));
```

```
    stream += sizeof(uint32_t);
    persona->nombre = malloc(persona->nombre_length);
    memcpy(persona->nombre, stream, persona->nombre_length);

    return persona;
}
```

¡Listo! Ya tenemos nuestro mensaje deserializado en el receptor listo para usar.

Notar que el desplazamiento lo fuimos sumando directamente a la variable stream, esto es completamente válido y produce el mismo resultado que el ejemplo de antes.

Attribute packed (gcc)

`GCC` , el compilador que utilizaremos para la realización del trabajo práctico, se le puede agregar al final de nuestro TAD, la siguiente anotación: `__attribute__((packed))` .

¿Por qué? De aquella forma le decimos a nuestro compilador que no se le agregue padding.

Sin embargo, hay que tener en cuenta que si no tenemos la estructura definida en un lugar único y decidimos invertir el orden de los atributos del struct (es decir, el emisor lo tiene de una forma y el receptor de otra) el compilador respetará el orden, lo cual provocará que los datos que le lleguen sean cualquier cosa, dado que tienen un orden distinto. Entonces, si bien dicha anotación nos facilita el envío de nuestra estructura, debemos tener cuidado con la misma.

c

```
typedef struct {
    uint32_t dni;
    uint8_t edad;
    uint32_t pasaporte;
       char nombre[14];
} __attribute__((packed))
t_persona;
```

Notas finales

¡Y eso fue todo! Les recordamos que cualquier duda que se les presente, la pueden realizar en los **medios de consulta** correspondientes, y que hagan uso de las guías y los video tutoriales de esta página.