

Práctica de Organización del Computador II

Clase pre-parcial

Segundo Cuatrimestre 2025

Arquitectura y Organización del computador (ex-ORGA2)
DC - UBA

Objetivos de la clase de hoy:

- Repaso sobre offsets, alineación de estructuras.
- Repaso sobre cómo recorrer estructuras.
- Repaso sobre ABI
- Repaso sobre llamados a funciones

Antes de comenzar: Aclaraciones



Antes de comenzar: Aclaraciones



Enunciado

La cátedra continúa trabajando en su juego de cartas Ah-Yi-Ok!, en el cual se enfrentan dos jugadores. En este juego, cada jugador cuenta con una mano de cartas que irán colocando en un tablero de 10x5 espacios para activar las distintas acciones asociadas a las cartas.



```
#define ANCHO_CAMPO 10  
#define ALTO_CAMPO 5
```

Enunciado



Cada carta tiene un nombre, un dueño y una cantidad de puntos de vida. Una carta puede estar en el campo de juego pero no estar en juego aún (se encuentra desactivada temporalmente o nunca se activó).

```
typedef struct carta {  
    bool en_juego;  
    char nombre[12];  
    uint16_t vida;  
    uint8_t jugador;  
} carta_t;
```

Enunciado

Los jugadores pueden decidir utilizar distintas acciones entre las de sus cartas en juego. Cada acción tiene asociadas una pieza de código, una carta destino afectada por la acción, y una acción siguiente a ella. Los punteros nulos se interpretan como la acción "fin del turno".

```
typedef void accion_fn_t(tablero_t* tablero, carta_t* carta);

typedef struct accion {
    accion_fn_t* invocar;
    carta_t* destino;
    struct accion* siguiente;
} accion_t;
```

Enunciado

Participan del juego dos jugadores humanos: el jugador rojo y el jugador azul. Además, el sistema del juego puede incorporar jugadores simulados (no-humanos) adicionales que también poseen la capacidad de colocar cartas en el tablero.

```
#define JUGADOR_ROJO 1  
#define JUGADOR_AZUL 2
```

Cada jugador cuenta con una "mano" de cartas de las cuales colocar las que desee en el campo de juego. Los jugadores no-humanos no tienen una "mano".

```
typedef struct tablero {  
    carta_t* mano_jugador_rojo;  
    carta_t* mano_jugador_azul;  
    carta_t* campo[ALTO_CAMPO][ANCHO_CAMPO];  
} tablero_t;
```

ESPACIO DE CONSULTAS



PRIMER TIP



Antes de comenzar es fundamental comprender el enunciado correctamente. Este es el espacio donde más consultas surgen porque entender mal el ejercicio implica que en primera instancia comenzaremos a resolverlo mal. ¡PREGUNTEN!

Ejercicio 0 - Enunciado

Completar los offsets y tamaños de struct definidos.

```
carta.en_juego EQU NO_COMPLETADO
carta.nombre   EQU NO_COMPLETADO
carta.vida     EQU NO_COMPLETADO
carta.jugador  EQU NO_COMPLETADO
carta.SIZE      EQU NO_COMPLETADO

tablero.mano_jugador_rojo EQU NO_COMPLETADO
tablero.mano_jugador_azul EQU NO_COMPLETADO
tablero.campo             EQU NO_COMPLETADO
tablero.SIZE               EQU NO_COMPLETADO

accion.invocar    EQU NO_COMPLETADO
accion.destino    EQU NO_COMPLETADO
accion.siguiente  EQU NO_COMPLETADO
accion.SIZE       EQU NO_COMPLETADO
```

Ejercicio 0 - Offsets

```
typedef struct carta {
    bool en_juego;
    char nombre[12];
    uint16_t vida;
    uint8_t jugador;
} carta_t;

typedef struct tablero {
    carta_t* mano_jugador_rojo;
    carta_t* mano_jugador_azul;
    carta_t* campo[ALTO_CAMPO][ANCHO_CAMPO];
} tablero_t;

typedef void accion_fn_t(tablero_t* tablero,
    carta_t* carta);

typedef struct accion {
    accion_fn_t* invocar;
    carta_t* destino;
    struct accion* siguiente;
}
```

Ejercicio 0 - Offsets

- Como primer ejercicio nos piden completar los offsets y tamaños de las estructuras.
- Recordemos que los offsets corresponden al tamaño del espacio en memoria que se ocupa hasta cada campo de la estructura en cuestión.
- Para calcular los offsets es importante reconocer los tipos de datos con los que estaremos trabajando.

Empecemos con carta_t.

```
typedef struct carta {  
    bool en_juego;  
    char nombre[12];  
    uint16_t vida;  
    uint8_t jugador;  
} carta_t;
```

Ejercicio 0 - Offsets

- Como primer ejercicio nos piden completar los offsets y tamaños de las estructuras.
- Recordemos que los offsets corresponden al tamaño del espacio en memoria que ocupa hasta cada campo de la estructura en cuestión.
- Para calcular los offsets es importante reconocer los tipos de datos con los que estaremos trabajando.

Empecemos con carta_t.

```
typedef struct carta {  
    bool en_juego; //1 byte  
    char nombre[12]; //1 byte * 12  
    uint16_t vida; //2 byte  
    uint8_t jugador; //1 byte  
} carta_t; //ocupa 16 bytes en memoria???
```

Ejercicio 0 - Offsets

Logramos identificar los tamaños de los diferentes tipos de datos.
¿Esto significa que ya podemos completar los offsets pedidos?

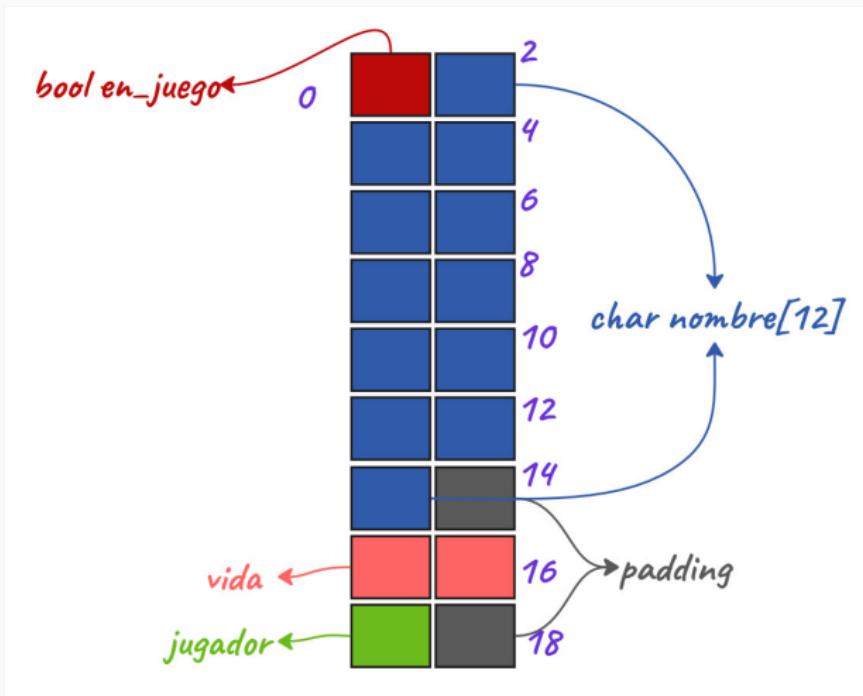
Ejercicio 0 - Offsets

Alineación:

- En estos momentos los gráficos se convierten en nuestro mejor aliado. Es muy importante graficar correctamente las estructuras ya que estos dibujos nos acompañarán durante el resto del parcial.
- Como siguiente paso debemos preguntarnos: ¿Es un struct packed? ¿Cómo se alinean los campos dentro de una estructura? ¿Cuál es el campo de mayor tamaño? ¿Todos los campos se alinean de igual manera? ¿Qué ocurre con un dato de dos bytes? ¿y con un bool?

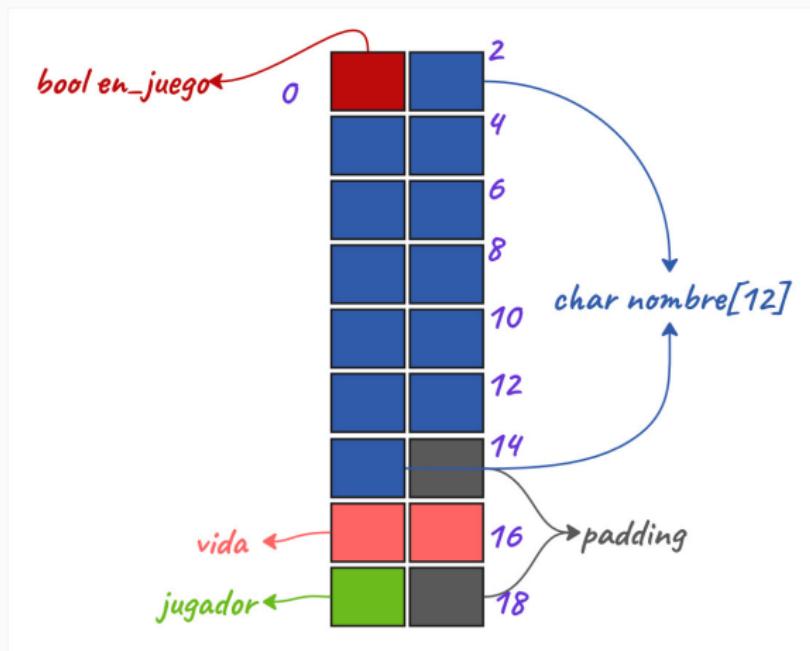
Ejercicio 0 - Offsets

Como no es un struct packed, el struct se alinea al tamaño del tipo más grande de todos. En este caso, se alinea a 2 bytes.



Ejercicio 0 - Offsets

Como se puede ver en la imagen, al tamaño total de los datos se le suma el padding, como el padding es 2 entonces el tamaño de cartas es de 18 bytes.



Ejercicio 0 - Offsets

Ahora tenemos que calcular los offsets de tablero_t.

```
tablero.ANCHO EQU 10
tablero.ALTO   EQU 5
typedef struct tablero {
    carta_t* mano_jugador_rojo;
    carta_t* mano_jugador_azul;
    carta_t* campo[ALTO_CAMPO][ANCHO_CAMPO];
} tablero_t;
```

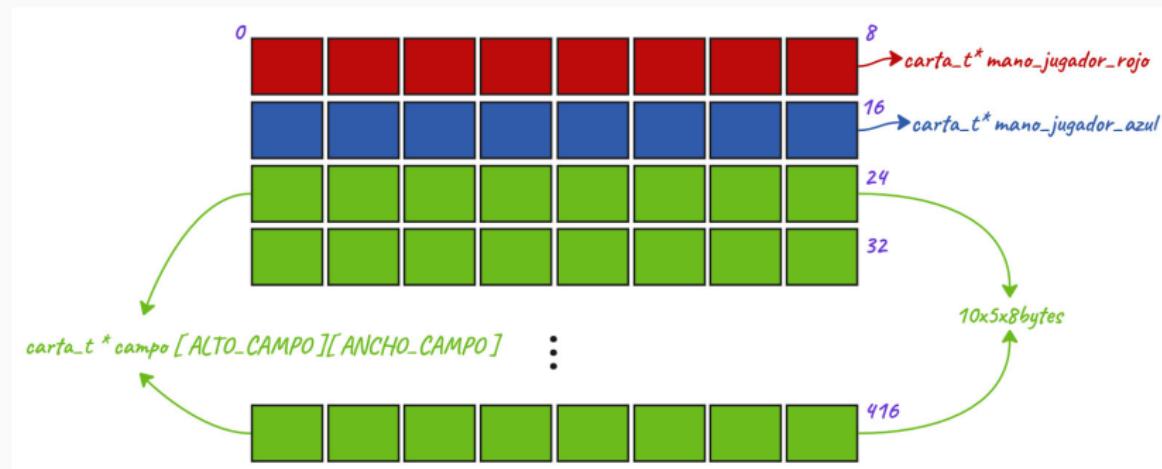
Ejercicio 0 - Offsets

Identificamos los tamaños de los tipos de datos.

```
tablero.ANCHO EQU 10
tablero.ALTO   EQU 5
typedef struct tablero {
    carta_t* mano_jugador_rojo; // 8 bytes
    carta_t* mano_jugador_azul; // 8 bytes
    carta_t* campo[ALTO_CAMPO][ANCHO_CAMPO];
                    // 8 bytes * 10 * 5
} tablero_t; //ocupa 416 bytes?
```

Ejercicio 0 - Offsets

Dado que el campo de mayor tamaño es 8 bytes, la alineación va a ser a 8 bytes y como todos los campos son de 8 bytes, no se va a necesitar padding.



Ejercicio 0 - Offsets

Por ultimo, nos queda definir los offsets de accion_t.

```
typedef void accion_fn_t(tablero_t* tablero,
    carta_t* carta);
typedef struct accion {
    accion_fn_t* invocar;
    carta_t* destino;
    struct accion* siguiente;
} accion_t;
```

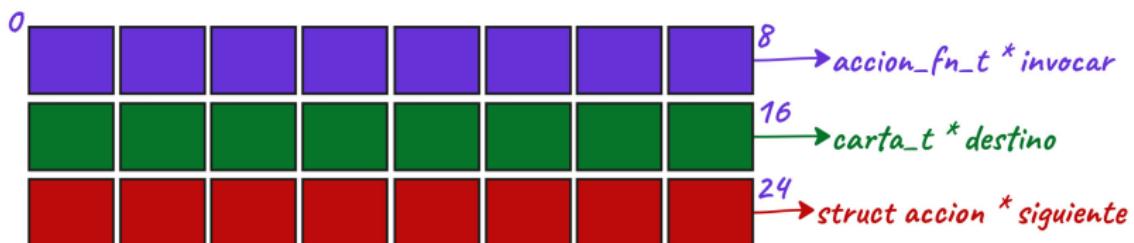
Ejercicio 0 - Offsets

Identificamos los tamaños de los tipos de datos.

```
typedef void accion_fn_t(tablero_t* tablero,
    carta_t* carta);
typedef struct accion {
    accion_fn_t* invocar; // 8 bytes
    carta_t* destino;   // 8 bytes
    struct accion* siguiente; // 8 bytes
} accion_t; // ocupa 24 bytes?
```

Ejercicio 0 - Offsets

Esta estructura es bastante parecida a la anterior, está formada por punteros de 8 bytes, por lo que se alinea a 8 bytes y no se va a necesitar padding. En memoria se podría ver a `accion_t` de esta forma:



Ejercicio 0 - Offsets - Solución

```
carta.en_juego EQU 0
carta.nombre    EQU 1
carta.vida      EQU 14
carta.jugador   EQU 16
carta.SIZE      EQU 18

tablero.mano_jugador_rojo EQU 0
tablero.mano_jugador_azul EQU 8
tablero.campo             EQU 16
tablero.SIZE              EQU 416

accion.invocar   EQU 0
accion.destino   EQU 8
accion.siguiente EQU 16
accion.SIZE      EQU 24
```

ESPACIO DE CONSULTAS



Ejercicio 1 - hay_accion_que_toque - Enunciado

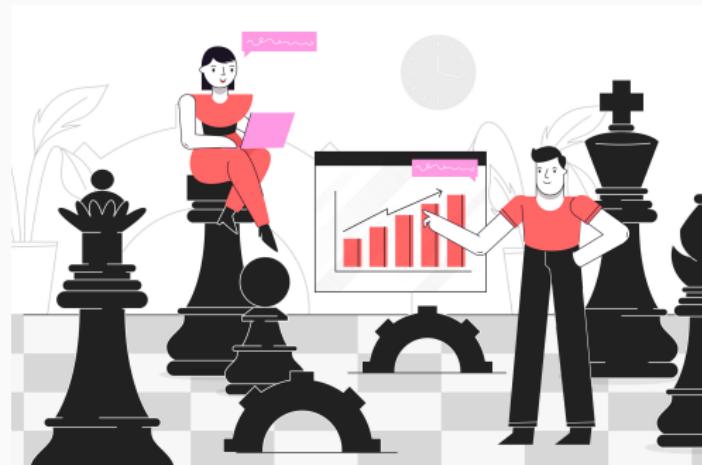
Dada una secuencia de acciones determinar si hay alguna cuya carta tenga un nombre idéntico (mismos contenidos, no mismo puntero) al pasado por parámetro.

```
bool hay_accion_que_toque(accion_t* accion,  
                           char* nombre);
```

El resultado es un valor booleano, la representación de los booleanos de C es la siguiente:

- El valor 0 es false.
- Cualquier otro valor es true.

SEGUNDO TIP

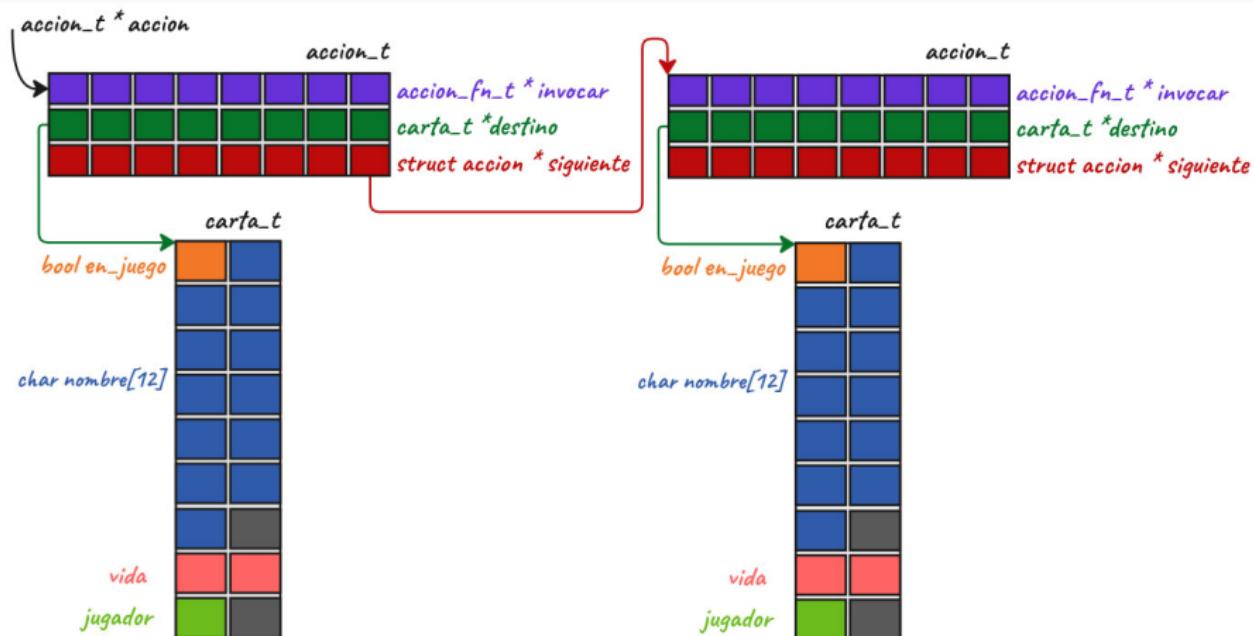


Plantear una estrategia. Es una buena idea analizar y chequear las estructuras con las que contamos y el objetivo a cumplir que nos piden. Meternos a codear de una a veces puede desencadenar que nos olvidemos cosas importantes en el camino.

¿Qué estrategia abordamos?

- Recorremos la lista de acciones.
- En cada acción visitada debo acceder a la carta destino.
- En cada carta visitada debo comparar su nombre con el que me pasan por parámetro. ¿Cómo lo hacemos?
- Si encuentro nombres iguales directamente devuelvo true. Si recorri toda la secuencia y no encontré nada devuelvo false.

Ideas



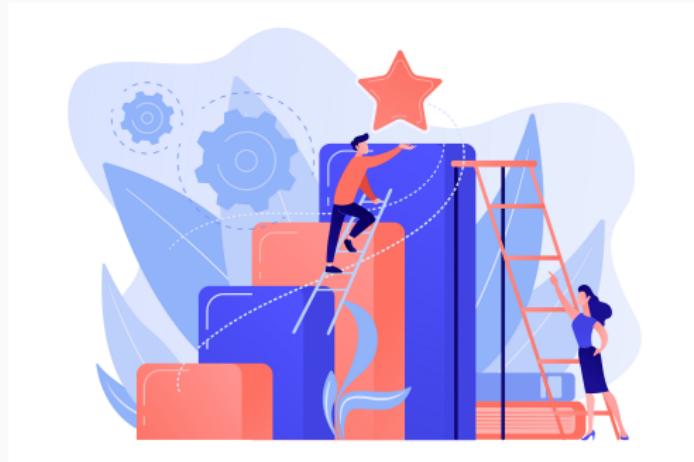
Ejercicio 1 - solución en C

```
bool hay_accion_que_toque(accion_t* accion, char* nombre) {
    bool hayAccion=false;
    int resultadoDeComparacion;
    while(accion!=NULL){ //recorro la lista de acciones buscando una que afecte a la carta
        resultadoDeComparacion = strcmp(accion->destino->nombre,nombre); //comparo el nombre
        if (resultadoDeComparacion == 0){//si son iguales corto el while y devuelvo true
            hayAccion=true;
            break; //Termino el while porque ya encontre una accion
        }
        accion = accion->siguiente; //siguiente elemento de la lista
    }
    return hayAccion; //devuelvo el resultado
}
```

Ejercicio 1 - solución en ASM

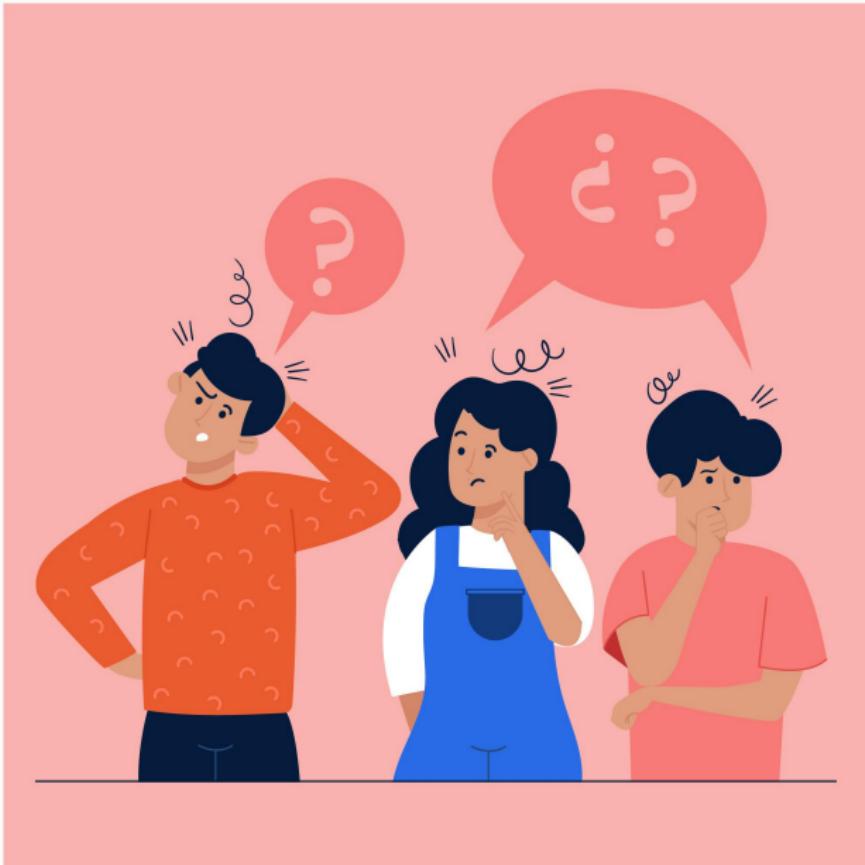
```
hay_accion_que_toque:  
    ; rdi = accion_t*  accion  
    ; rsi = char*      nombre  
    push rbp  
    mov rbp, rsp  
    push r12  
    push r13  
    push r14 ;prologo  
    sub rsp, 8 ;alineacion de pila  
    xor rax, rax ;rax = 0  
    mov r12, rdi ;r12 = puntero a accion  
    mov r13, rsi ;r13 = puntero al nombre  
    mov r14, FALSE ;r14 = false  
.ciclo:  
    cmp r12, 0 ; puntero a null, terminó el ciclo  
    je .fin  
    mov rdi, [r12 + accion.destino] ; rdi = puntero a carta destino  
    lea rdi, [rdi + carta.nombre] ; rdi = puntero al nombre de la carta  
    mov rsi, r13 ; rsi = puntero al nombre de carta buscado  
    call strcmp ; llamo a strcmp  
    cmp eax, 0 ; son iguales  
    jne .siguiente ; si no son iguales se salta a siguiente  
    mov r14, TRUE ; r14 = true  
    jmp .fin ;break  
.siguiente:  
    mov r12, [r12 + accion.siguiiente] ; r12 = siguiente accion  
    jmp .ciclo ; salto al ciclo  
.fin:  
    mov rax, r14 ; devuelvo el resultado en rax  
    add rsp, 8  
    pop r14 ; epilogo  
    pop r13  
    pop r12  
    pop rbp  
    ret
```

TERCER TIP



Escribir código es como construir un edificio, es mas difícil encontrar y corregir un error al final que al principio. Encontrar un error cuando hay menos código cuesta menos trabajo, se puede aprovechar los test y la inmediatez de estos para confirmar que no nos estamos mandando una macana.

ESPACIO DE CONSULTAS



Ejercicio 2 - invocar_acciones - Enunciado

Se pide implementar `invocar_acciones`, que ejecute una secuencia de acciones sobre un tablero en el orden dado.

```
void invocar_acciones(accion_t* accion,  
                      tablero_t* tablero);
```

Una acción sólo se ejecuta si la carta destino está en juego.

Al ejecutarse, se llama a la función de acción con (`tablero`, `carta_destino`).

```
void mi_accion(tablero_t* tablero, carta_t*  
                carta);
```

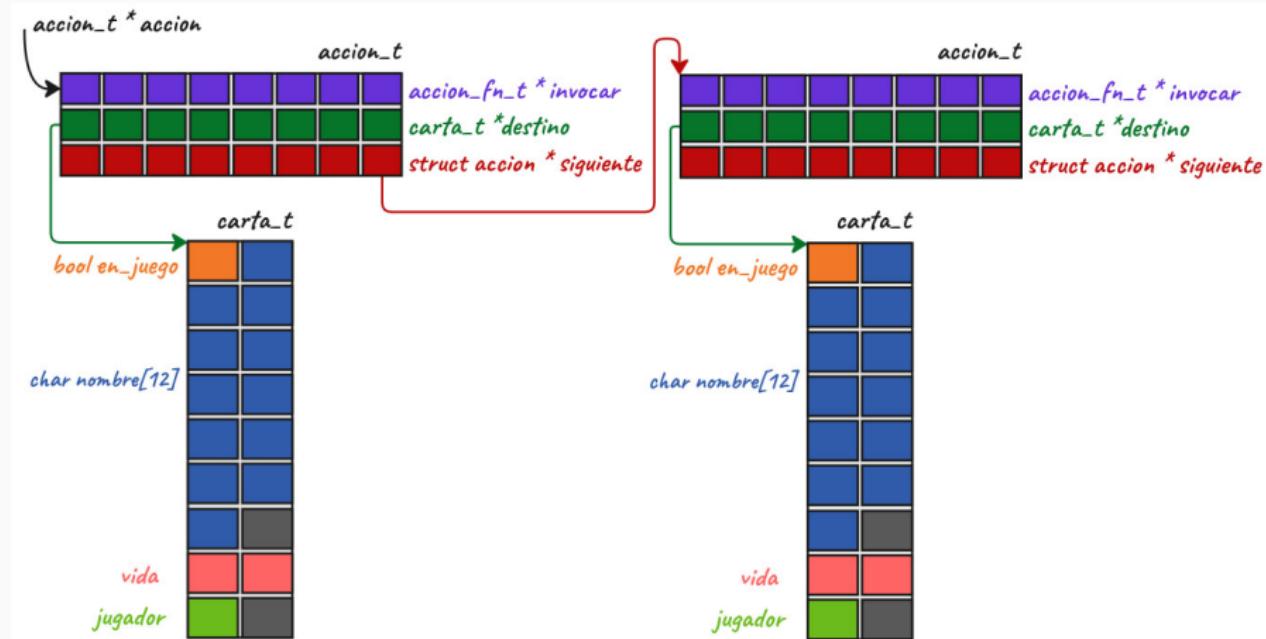
Si la carta destino queda con 0 de vida después de la acción, pasa a estar fuera de juego.

Antes de la acción una carta con 0 de vida puede seguir en juego, y también puede salir de juego por otros motivos (no solo por vida).

¿Qué estrategia abordamos?

- Recorremos la lista de acciones.
- En cada acción visitada debo chequear si la carta está en juego.
- Si está en juego, ejecutamos la acción llamando a la función del campo **invocar** . ¿Cómo llamamos a esa función? ¿Recibe parámetros? ¿Cómo se los pasamos?
- Luego de invocar, chequeamos la vida de la carta. Si se quedó sin vida debemos sacarla del juego.

Ideas



Ejercicio 2 - solución en C

```
void invocar_acciones(accion_t* accion, tablero_t* tablero) {
    carta_t* cartaDestino; //defino variable
    while(accion!=NULL){ //recorro toda la lista de acciones
        cartaDestino = accion->destino; //defino la variable
        if(cartaDestino->en_juego){ //si la carta está en juego la invoco
            accion->invocar(tablero, cartaDestino); //Invoco la accion llamando a la funcion
            if(cartaDestino -> vida ==0){//si la vida de la carta quedó en 0 la saco del juego
                cartaDestino -> en_juego=false;
            }
        }
        accion=accion -> siguiente;
    }
}
```

Ejercicio 2 - solución en ASM

```
global invocar_acciones
invocar_acciones:
    ; rdi = accion_t* accion
    ; rsi = tablero_t* tablero
    push rbp
    mov rbp, rsp
    push r12
    push r13 ;epilogo
    mov r12, rdi ;lista de acciones
    mov r13, rsi ;tablero
.ciclo:
    cmp r12, 0 ; comparo r12 con nulo
    je .fin ; si es nulo salto a fin
    mov r8, [r12+accion.destino] ;r8=puntero a carta
    cmp BYTE [r8+carta.en_juego] , FALSE ; veo si la carta esta en juego
    je .siguiente ;carta en juego
    mov rdi, r13 ;puntero a tablero
    mov rsi, r8 ;puntero a carta
    call [r12+accion.invocar] ;llamo a funcion
    mov r8, [r12+accion.destino] ;r8=puntero a carta
    cmp WORD[r8 + carta.vida], 0 ;comparo la vida de la carta con 0
    jne .siguiente ; est  a viva
    mov BYTE[r8 + carta.en_juego], 0 ; deja de estar en juego
.siguiente
    mov r12, [r12 + accion.siguiente] ; siguiente accion
    jmp invocar_acciones.ciclo ;vuelvo al ciclo
.fin:
    pop r13 ;epilogo
    pop r12
    pop rbp
    ret
```

ESPACIO DE CONSULTAS



Ejercicio 3 - contar_cartas - Enunciado

Contar la cantidad de cartas en el tablero correspondientes a cada uno de los jugadores.

```
void contar_cartas(tablero_t* tablero ,  
                    uint32_t* cant_rojas , uint32_t*  
                    cant_azules);
```

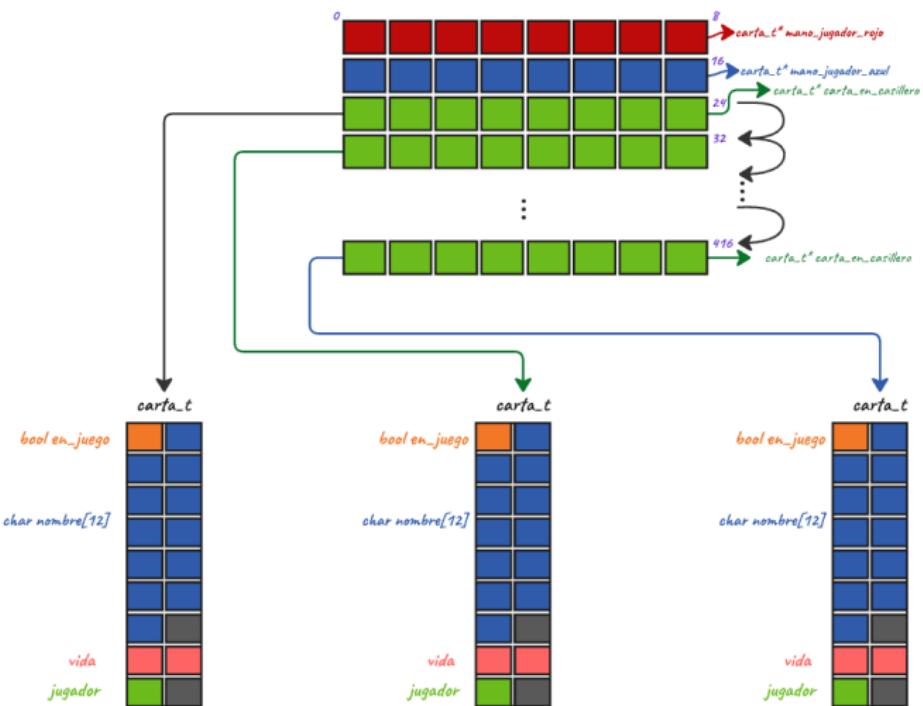
Se deben tener en cuenta las siguientes consideraciones:

Además del jugador rojo y el jugador azul puede haber cartas asociadas a otros jugadores simulados (no-humanos). Las posiciones libres del campo tienen punteros nulos en lugar de apuntar a una carta. El resultado debe ser escrito en las posiciones de memoria proporcionadas como parámetro. El conteo incluye tanto a las cartas en juego como a las fuera de juego (siempre que estén visibles en el campo).

¿Qué estrategia abordamos?

- Recorremos el campo, como una matriz. ¿Podemos reinterpretar esta estructura para recorrerla mas secillamente?.
- En cada casilla nos fijamos si hay una carta o no.
- Si hay una carta, nos fijamos si es del jugador rojo o del azul, teniendo cuidado porque pueden existir otros jugadores.
- Y si es del rojo o del azul, lo sumamos al total de sus cartas.

Ideas



Ejercicio 3 - solución en C

```
void contar_cartas(tablero_t* tablero, uint32_t* cant_rojas, uint32_t* cant_azules) {
    *cant_rojas = *cant_azules = 0; //escribo 0 en los resultados
    carta_t* cartaEnCasilla;
    for(int i=0;i<ALTO_CAMPO;i++){//recorro todas las columnas
        for(int j=0;j<ANCHO_CAMPO;j++){//recorro todas las filas
            cartaEnCasilla = tablero->campo[i][j];//agarró carta en la casilla
            if (cartaEnCasilla==NULL){//si es puntero a null salto a la siguiente casilla
                continue;
            }
            if (cartaEnCasilla->jugador==JUGADOR_AZUL){//si es del jugador azul le sumo 1
                *cant_azules+=1;
            }
            if (cartaEnCasilla->jugador==JUGADOR_ROJO){//si es del jugador rojo le sumo 1
                *cant_rojas+=1;
            }
        }
    }
}
```

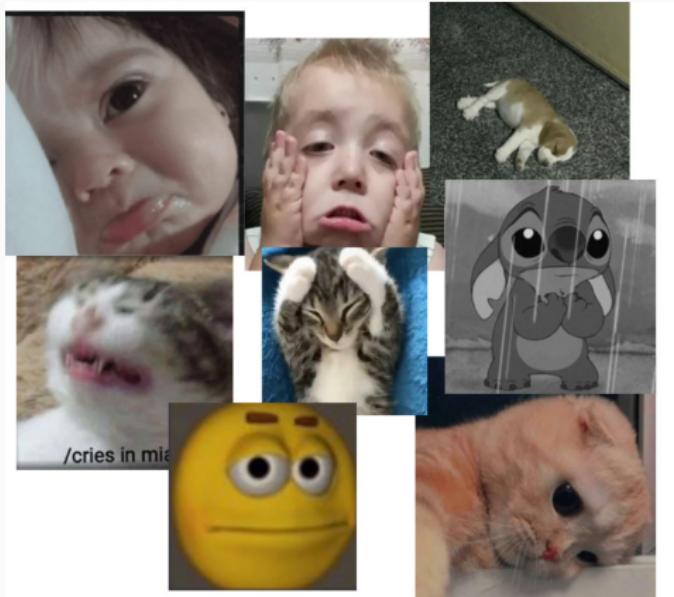
Ejercicio 3 - solución en ASM

```
contar_cartas:  
    ; rdi = tablero_t* tablero  
    ; rsi = uint32_t* cant_rojas  
    ; rdx = uint32_t* cant_azules  
    push rbp  
    mov rbp,esp ;prologo  
    lea rdi, [rdi + tablero.campo] ;rdi = puntero al primer casillero  
    mov DWORD[rsi], 0 ; *cant_rojas = 0  
    mov DWORD[rdx], 0 ; *cant_azules = 0  
    xor r9,r9 ;r 9=0  
.ciclo:  
    cmp r9, tablero.ALTO*tablero.ANCHO ;veo si recorri todos los casilleros  
    je .fin ; salto al final si es true  
    mov r10, [rdi + r9*8] ;R10 = cartaEnCasilla  
    cmp r10, 0 ; comparo r10 con null  
    je .siguiente ; si es null salto al siguiente  
.jugadorAzul:  
    cmp BYTE [r10 + carta.jugador], JUGADOR_AZUL ;veo si es una carta del jugador azul  
    jne .jugadorRojo  
    inc DWORD[rdx] ;incremento el contador de las azules  
    jmp .siguiente  
.jugadorRojo:  
    cmp BYTE [r10 + carta.jugador], JUGADOR_ROJO ;veo si es una carta del jugador rojo  
    jne .siguiente  
    inc DWORD[rsi] ;incremento el contador de las rojas  
.siguiente:  
    inc r9 ; sumo uno al contador de casilleros  
    jmp .ciclo ; salto al ciclo  
.fin:  
    pop rbp ;epilogo  
    ret
```

ESPACIO DE CONSULTAS



ULTIMO TIP



¡No entren en pánico! Nuestra mente trabaja mejor cuando estamos tranquilos.

ULTIMO TIP



El examen es un paso más en el camino, no el destino final. Lo importante es cuánto crecieron y todo lo que pudieron aprender. Respiren, confíen, y recuerden que saben mucho más de lo que creen.