

## Recuperatorio Primer Parcial

### Primer Cuatrimestre 2024

### Normas generales

- El parcial es INDIVIDUAL
- Una vez terminada la evaluación se deberá completar un formulario con el *hash* del *commit* del repositorio de entrega. El link al mismo es: <https://forms.gle/pTwNZLtatMi6Ym9A>.
- De ser necesario, luego de la entrega habrá una instancia coloquial de defensa del trabajo

### Régimen de Aprobación

- Para aprobar el examen es necesario obtener como mínimo **60 puntos**.

### Compilación y Testeo

Para compilar y ejecutar los tests se dispone de un archivo `Makefile` con los siguientes *targets*

<code>make test_c</code>	Genera el ejecutable usando la implementación en C del ejercicio.
<code>make test_asm</code>	Genera el ejecutable usando la implementación en ASM del ejercicio.
<code>make run_c</code>	Corre los tests usando la implementación en C.
<code>make run_asm</code>	Corre los tests usando la implementación en ASM.
<code>make valgrind_c</code>	Corre los tests en valgrind usando la implementación en C.
<code>make valgrind_asm</code>	Corre los tests en valgrind usando la implementación en ASM.

Además, tener en cuenta lo siguiente:

- Cada ejercicio tiene su propio `Makefile` con los targets arriba mencionados
- Es importante que lean la documentación provista en el archivo header correspondiente (ej1.h o ej2.h según corresponda). La documentación completa de las funciones se encuentra en dichos headers, no en el presente enunciado.
- Todos los incisos del parcial son independientes entre sí.
- El sistema de tests del parcial sólo correrá los tests que hayan marcado como hechos. Para esto deben modificar la variable `EJERCICIO_xx_HECHO` correspondiente asignándole `true` (en C) ó `TRUE` (en ASM). `xx` es el inciso en cuestión: 1A, 1B, 1C, 2A o 2B.

## Ej. 1 - (50 puntos)

En este ejercicio vamos a explorar un *framework* sencillo para manejar cadenas de caracteres el cual nos va a permitir combinar en forma arbitraria cadenas simples para formar cadenas de caracteres más complejas.

Las cadenas se representan haciendo uso de tres tipos `texto_literal_t`, `texto_concatenacion_t` y `texto_cualquiera_t`.

El primero, `texto_literal_t` representa un string sencillo:

```
typedef struct {
    uint32_t tipo;
    uint32_t usos;
    uint64_t tamano;
    const char* contenido;
} texto_literal_t;
```

---

tipo	Siempre es el valor numérico <code>TEXTO_LITERAL</code> (Ver defines en <code>ej1.h</code> ).
usos	La cantidad de otras cadenas que usan esta cadena. Al momento de su creación este valor es 0.
tamano	La cantidad de caracteres de la cadena (sin contar el carácter nulo de terminación).
contenido	La cadena en sí (representada como una <i>C string</i> clásica).

---

El segundo, `texto_concatenacion_t` representa una concatenación de dos cadenas:

```
typedef struct {
    uint32_t tipo;
    uint32_t usos;
    uint64_t tamano;
    texto_cualquiera_t* izquierda;
    texto_cualquiera_t* derecha;
} texto_concatenacion_t;
```

---

tipo	Siempre es el valor numérico <code>TEXTO_CONCATENACION</code> (Ver defines en <code>ej1.h</code> ).
usos	La cantidad de otras cadenas que usan esta cadena. Al momento de su creación este valor es 0.
tamano	La longitud del texto concatenado
izquierda	La cadena de la izquierda. Puede referenciar a un <code>texto_literal_t</code> o a un <code>texto_concatenacion_t</code> .
derecha	La cadena de la derecha. Puede referenciar a un <code>texto_literal_t</code> o a un <code>texto_concatenacion_t</code> .

---

Nótese que el tipo `texto_concatenacion_t` **no contiene las cadenas en sí**, sino que mantiene solo las referencias. Además, se puede asumir que el tamaño de los textos 'izquierda' y 'derecha' está bien calculado (para usarlos al calcular el tamaño del texto a crear).

Finalmente, `texto_cualquiera_t` sirve para poder referirse a cualquiera de los dos tipos anteriores de manera genérica:

```
typedef struct {
    uint32_t tipo;
    uint32_t usos;
    uint64_t tamano;
} texto_cualquiera_t;
```

A continuación algunos ejemplos gráficos de como se podrían usar estos tipos de datos. Las estructuras marcadas en **rojo** representan `texto_literal_t` y las marcadas en negro son `texto_concatenacion_t`.

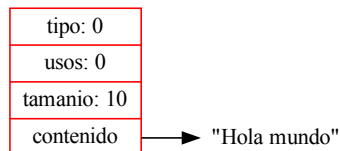


Figura 1: Ejemplo de un texto constituido por un único literal. Representa la cadena `Hola mundo`.

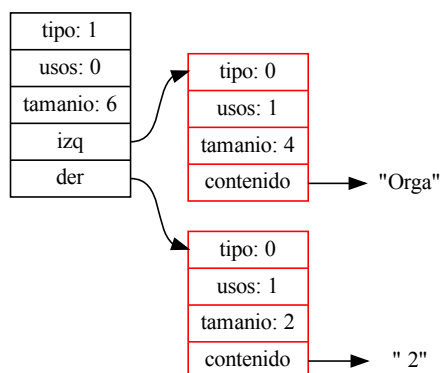


Figura 2: Ejemplo de un texto constituido por una concatenación de dos literales. Representa la cadena `Orga 2`. Nótese el espacio en el string de la derecha.



```

/* Imprime un texto */
void texto_imprimir(texto_cualquiera_t* texto);

/* Libera el texto pasado como parámetro */
bool texto_liberar(texto_cualquiera_t* texto);

```

### IMPORTANTE:

Para entender qué hace cada función lean la documentación completa en ej1.h

### A entregar:

Se pide realizar en assembly (en el archivo ej1.asm) las siguientes funciones:

- a) */\* Crea un `texto\_literal\_t` que representa la cadena pasada por parámetro. \*/*  
`texto_literal_t* texto_literal(const char* texto);`  
  
*/\* Crea un `texto\_concatenacion\_t` que representa la concatenación de ambos parámetros. \*/*  
`texto_concatenacion_t* texto_concatenar(texto_cualquiera_t* izquierda, texto_cualquiera_t* derecha);`
- b) */\* Calcula el tamaño total de un `texto\_cualquiera\_t` y todos sus nodos internos. Actualiza el campo `tamano` del texto pasado como parámetro. \*/*  
  
`void texto_calcular_tamano(texto_cualquiera_t* texto);`
- c) */\* Chequea si los tamaños de todos los nodos internos al parámetro corresponden al tamaño del texto representado. Es decir: si los campos `tamano` están bien calculados. \*/*  
`bool texto_chequear_tamano(texto_cualquiera_t* texto);`

Tanto el archivo ej1.c como ej1.asm tienen implementaciones de texto\_imprimir y texto\_liberar. Para que las implementaciones en ASM funcionen correctamente **deben completar las definiciones de sizes y offsets que figuran en ej1.asm.**

## Ej. 2 - (50 puntos)

Pese a sus exitosas ventas la *OrgaStation2* no es la consola más potente del mercado. Para obtener buena fidelidad gráfica muchos juegos decidieron pre-renderizar contenido 3D el cual mezclan con gráficos propios de la consola.

Hacer esto es relativamente fácil. Se toman dos imágenes en formato RGBD de las cuales se tiene su mapa de color (3 canales, RGB, 8 bits por canal, sin signo) y se incluye la información de profundidad en el canal D (1 canal, escala de grises, 8 bits, sin signo). Entonces, el formato de las imágenes es RGBD es el siguiente:

```
/**
 * Un píxel RGBD.
 *
 * Campos:
 * - r: La cantidad de rojo en el píxel. Va de 0 a 255.
 * - g: La cantidad de verde en el píxel. Va de 0 a 255.
 * - b: La cantidad de azul en el píxel. Va de 0 a 255.
 * - d: La profundidad del píxel. Va de 0 a 255.
 */
typedef struct rgbd_pixfmt {
    uint8_t r;
    uint8_t g;
    uint8_t b;
    uint8_t d;
} rgbd_t;
```

Por cada píxel en las imágenes fuente se elige el de **menor profundidad** y se escribe en la imagen destino. Es decir:

$$\text{dst}[y, x] = \begin{cases} a[y, x] & \text{si } \text{depth\_a}[y, x] < \text{depth\_b}[y, x] \\ b[y, x] & \text{sino} \end{cases}$$

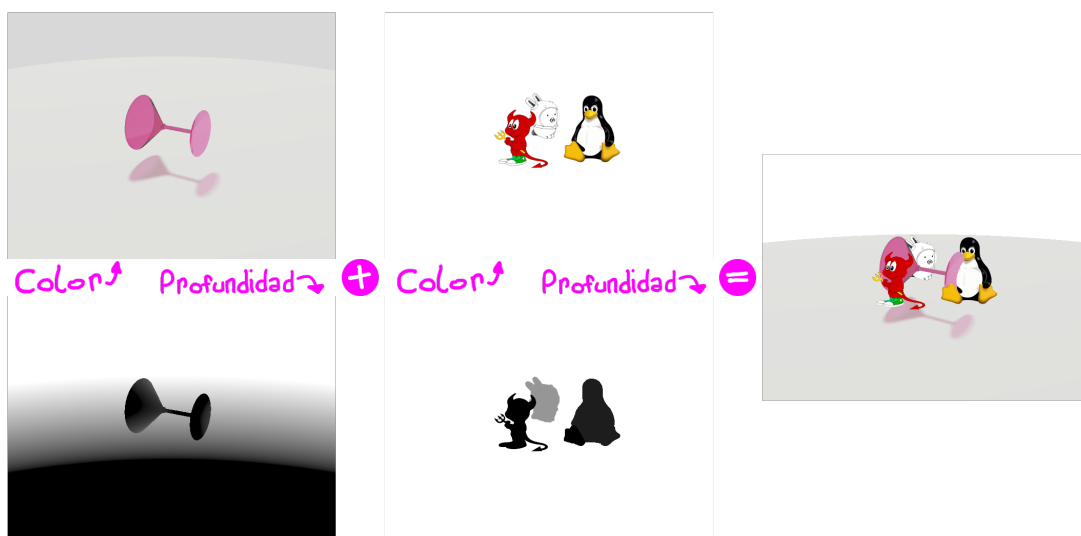


Figura 5: Una imagen compleja puede mezclarse con otra que es una serie de planos. Algunos planos quedan detrás de la imagen y otros por delante.



Figura 6: Dos escenas tridimensionales complejas pueden mezclarse. Los elementos cercanos a la cámara predominan en el resultado.

### A entregar:

En ASM, utilizando SIMD y teniendo como objetivo el procesamiento de 4 píxeles en simultáneo, se pide:

- a Implementar el algoritmo de “mezclado sensible a la profundidad”. De forma breve este algoritmo se puede describir como hacer  $dst[y, x] = \begin{cases} a[y, x] & \text{si } depth\_a[y, x] < depth\_b[y, x] \\ b[y, x] & \text{sino} \end{cases}$  en cada píxel de dst. En caso de empate entre las profundidades de A y B se debe escribir el píxel de B.

```
/**
 * Dadas dos imágenes de origen (`a` y `b`) en conjunto con sus mapas de
 * profundidad escribe en el destino el pixel de menor profundidad por cada
 * píxel de la imagen. En caso de empate se escribe el píxel de `b`.
 *
 * Parámetros:
 * - dst: La imagen destino. Está a color (RGBA) en 8 bits sin signo por
 * canal.
 * - a: La imagen origen A. Está a color + profundidad (RGBD) en 8 bits
 * sin signo por canal.
 * - b: La imagen origen B. Está a color + profundidad (RGBD) en 8 bits
 * sin signo por canal.
 * - width: El ancho en píxeles las imágenes parámetro.
 * - height: El alto en píxeles las imágenes parámetro.
 */
void ej2(rgba_t* dst, rgbd_t* a, rgbd_t* b, uint32_t width, uint32_t height);
```

### Se puede asumir:

- Tanto el ancho como el alto de las imágenes (en píxeles) es múltiplo de 16

- Las imágenes a mezclar tienen el mismo ancho y alto (en píxeles).
- Se debe escribir alpha de la imagen destino en 255.