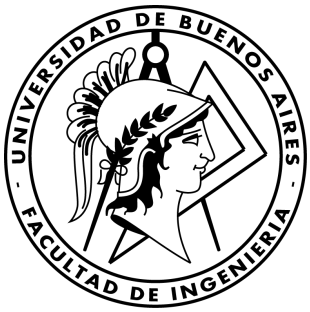


# Trabajo Práctico 01
# Sala de escape Pokemon

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

---


| **Autor**: | **Email**: | **Padrón**: | **Fecha**: |
|---|---|---|---|
| Luca Lazcano | llazcano@fi.uba.ar | 107044 | 10/04/2022 |

# Índice

# 1. Introducción

En este trabajo práctico, se pone al alumno como personaje de un juego. Se encuentra en una sala de la cual busca escapar y donde se encuentran ciertos objetos. Se les pueden realizar ciertas acciones (como leer una nota o abrir una pokebola), y pueden interactuar entre ellos de ciertas formas definidas (como usar una llave para abrir un cajón). Los objetos de la sala y las interacciones posibles se proveen en dos archivos de texto.

En el archivo de los objetos, para cada uno se detalla el nombre, una descripción y un indicador que puede ser verdadero o falso.

En el archivo de las interacciones, para cada objeto se indica en qué consiste la interacción, mediante un verbo; de ser necesario para la interacción, un segundo objeto; y un campo acción.

Este campo acción a su vez consiste de tres campos: el tipo de interacción, limitada a cuatro posibilidades (descubrir, reemplazar o eliminar el objeto, o mostrar un mensaje); el nombre de otro objeto resultante de la acción (si lo hubiese); y un mensaje.

Provistos los archivos, ciertas estructuras de datos, y literales de algunas funciones, se pide implementar la lectura de los archivos para la correcta creación de la sala de escape. Se deben listar por pantalla todos los objetos leídos del archivo. Dadas algunas interacciones entre objetos, se debe determinar su validez.

Para la implementación se deben respetar algunas directivas, que son: recibir por parámetro del `main` la ruta de los dos archivos y liberar toda la memoria utilizada durante la implementación. No se debe incluir el archivo `src/estructuras.h` provisto, ya que se considera privado de la implementación.

# 2. Compilación y ejecución

Una vez escrito el código fuente, es necesario utilizar un compilador para traducirlo a lenguaje máquina y generar un archivo de ejecución. Para la compilación se utiliza GCC (GNU Compiler Collection), un compilador desarrollado por The GNU Project y distribuido por la Free Software Foundation. Se especifican varias opciones de compila-

ción para personalizar la detección de errores y advertencias.

Para compilar el programa se debe ejecutar la siguiente línea de código desde la consola, en el directorio donde se encuentra el archivo:

```
gcc escape_pokemon.c src/*.c -g -O0 -std=c99 -Wall -Wconversion
↪  -Wtype-limits -Werror
```

Para ejecutar el programa abrimos el archivo ejecutable desde la consola, agregando los nombres de los archivos necesarios para la implementación del trabajo:

```
./a.out ejemplo/objetos.txt ejemplo/interacciones.txt
```

Además, podemos utilizar alguna herramienta para verificar el manejo de la memoria por parte de nuestro programa, y así detectar y localizar pérdidas o *leaks* de memoria, ayudando con la depuración o *debugging* del código. Para esto utilizamos Valgrind, una herramienta desarrollada por Julian Seward, bajo licencia GNU.

Para ejecutar el programa con Valgrind corremos:

```
valgrind ./a.out ejemplo/objetos.txt ejemplo/interacciones.txt
```

# 3. Lectura de archivos

La utilización de archivos tiene varias ventajas entre las que se destacan:

- Persistencia en el tiempo de la información. Los archivos son información que se guardan en el almacenamiento del dispositivo, por lo cual persisten aún cuando se prende y apaga el dispositivo, o dejan de ser utilizados por un programa.
- Conveniencia. Trabajar con archivos evita que debamos cargar manualmente la información cada vez que ejecutamos el programa. Pueden ser accedidos en cualquier momento desde el programa con algunos simples comandos. Cuanta más

información tengamos más se justifica la utilización de archivos. Además, es sencillo mover información de un dispositivo a otro para ser utilizado por otros programas, quedando así desligado de un programa en particular.

Para la manipulación de archivos en C se utilizan una serie de funciones incluidas en la biblioteca estándar `stdio.h`. Para trabajar con un archivo, en primer lugar se debe abrir, con la función `fopen()`, la cual devuelve un puntero de tipo `FILE`.

```
FILE *fopen( const char *filename, const char *mode );
```

Entonces se abre o crea el archivo apuntado por `filename` en modo `mode`. Los modos pueden ser ‘‘`r`’’: sólo lectura, ‘‘`w`’’: crea o sobreescribe el archivo para escribir, ‘‘`a`’’: para escribir al final del archivo, ‘‘`r`’’: abre el archivo para lectura y sobreescritura, ‘‘`w`’’: crea el archivo para lectura y sobreescritura, ‘‘`a`’’: abre el archivo para lectura y escritura al final del archivo.

La función devuelve `NULL` en caso de error, por lo que siempre se debe revisar que se haya podido abrir correctamente el archivo.

Entonces, para abrir los archivos de objetos e interacciones:

```
1 FILE *archivo_objetos = fopen( objetos, "r" );
2 FILE *archivo_interacciones = fopen( interacciones, "r" );
```

Funciones como `scanf()` o `fgets()` nos permiten leer el archivo línea por línea.

```
1 char *fgets( char *str, int n, FILE *stream );
```

Esta función lee línea por línea del archivo `stream`, devuelve el `str`, o `NULL` en caso de alcanzar `n-1` líneas o `EOF` (final del archivo). Es necesario revisar que no devuelva `NULL` para asegurarnos de leer correctamente la información dentro del archivo.

La lectura de los archivos fue implementada utilizando ciclos `while`.

```
#define MAX_LINEA 1024
...
    while( fgets( linea, MAX_LINEA, archivo_objetos ) != NULL ) {
        ...
    }
    while( fgets( linea, MAX_LINEA, archivo_interacciones ) != NULL ) {
        ...
    }
```

Siempre se debe cerrar el archivo en el programa al terminar de utilizarlo, para evitar corrupciones del archivo y otros problemas. Para ello se utiliza:

```
int fclose(FILE *stream)
```

Cierra el archivo `stream` y devuelve 0, o `EOF` en caso de error.

Esto fue implementado:

```
fclose( archivo_objetos );
fclose( archivo_interacciones );
```

# 4. Memoria dinámica

La memoria dinámica es aquella que se asigna del `heap` durante la ejecución del programa. La reserva de memoria dinámica es útil cuando no se conoce el tamaño de la información requerida. La vida de un bloque de memoria dinámica comienza cuando el programador la reserva del `heap` según sus requerimientos, y termina cuando es desasignada por el propio programador. Es necesario ser cauteloso en el manejo de memoria dinámica, so peligro de perder definitivamente acceso a bloques de memoria.

Para la asignación y desasignaciónde memoria se cuenta con varias funciones en la

librería estándar. Se destacan:

- `malloc()`: Asigna un tamaño de `size` bytes y devuelve un puntero al inicio del bloque de memoria reservada, o `NULL` en caso de error.

```
malloc( size_t size );
```

- `calloc()`: Asigna memoria para `items` elementos de tamaño `size` bytes, inicializa todos los elementos en 0, y devuelve un puntero al inicio del bloque de memoria reservada, o `NULL` en caso de error.

```
calloc( size_t nitems, size_t size );
```

- `realloc()`: Modifica a `size` el tamaño del bloque de memoria a la cual apunta `ptr`, y devuelve un puntero al inicio del bloque de memoria reservada, o `NULL` en caso de error.

```
realloc(void *ptr, size_t size);
```

- `realloc()`: Libera el bloque de memoria apuntado por `ptr`, que fue previamente reservado con alguna de las funciones vistas. Toda asignación de memoria necesita ser liberado, so pena de perder memoria disponible para la ejecución del programa.

```
free(void *ptr)
```

Es conveniente la utilización del operador `sizeof()` para determinar el tamaño de los datos para los cuales se requiere memoria.

Es importante verificar cada vez que intente asignar memoria, que se haya asignado correctamente, verificando que la función utilizada no devuelva `NULL`.

La memoria dinámica es manejada exclusivamente por el programador, por lo que es importante verificar la correcta asignación, no intentar escribir o leer datos más allá de la memoria asignada, y correctamente liberar toda la memoria utilizada antes de finalizar la ejecución del programa.

Como ya se comentó, Valgrind es una herramiento útil para detectar fugas de memoria.

# 5. Detalles de implementación

Recibidos como parámetros las direcciones de los archivos al `main` de `escape_pokemon.c`, se revisa que efectivamente se el número de argumentos sea el correspondiente (el nombre del archivo, por defecto, más los dos archivos). Caso contrario se devuelve -1.

Luego creamos un puntero de tipo `sala_t` y creamos la sala con la función `sala_crear_desde_ar` que crea la sala dados los archivos.

Si la sala fue correctamente creada, invocamos la función `sala_obtener_nombre_objetos`, con la sala creada y el puntero a una variable.

Esta función devuelve un puntero a un vector con los nombres de los objetos de la sala, los cuales podemos a continuación listar por pantalla.

Para las interacciones se evalúan distintos casos utilizando `sala_es_interaccion_valida` invocada con la sala, el verbo de la interacción, el objeto en cuestión, y eventualmente un segundo objeto. Se imprimen por pantalla las interacciones y la validez.

Finalmente, se destruye la sala, es decir, se libera toda la memoria asignada para sala y sus campos, con `sala_destruir`.

## 5.1. `sala_crear_desde_archivos`

En esta función se abren los archivos recibidos del `main`, se abren en modo lectura. Luego asignamos memoria para crear el objeto sala, con tamaño de la estructura `sala_t`.

A continuación se inicializa un vector de tipo `struct objeto` en `NULL`. En un ciclo `while` se parsea cada línea del archivo de objetos (mientras no se haya encontra el `EOF`), se asigna memoria para la creación de un bloque de objetos, o se reasigna de ser necesario. A cada posición de este objeto se le asigna un objeto creado desde la línea parseada, con `objeto_crear_desde_string`.

Finalizado el archivo, se le asigna al campo `objetos` de la sala el bloque de objetos, y su tamaño.

La creación de las interacciones se realiza de modo similar, inicializando un `struct interaccion`, parseando las líneas del archivo de interacciones y creando las interacciones con `interaccion_crear_desde_string`. Luego se asigna el bloque de interacciones al campo `interacciones` de la sala, y su tamaño.

Se cierran los archivos.

### 5.2. `interaccion_crear_desde_string`

Se asigna memoria para un `struct interaccion`. Luego se parsea la línea con `sscanf()` separando sus distintos campos y asignándolos al campo correspondiente de la estructura. Si el objeto parámetro es guión bajo, se guarda como un string vacío.

Para crear el campo acción, se asigna memoria, y en la función accion_crear_desde_string se crea el struct acción, de forma similar a la presente función. Se asigna al campo `accion` de la interacción, la acción creada y se libera le la memoria asignada.

Finalmente se devuelve el puntero a la interacción.

## 6. Diagramas

Se provee un diagrama de la memoria, con el `stack` y el `heap`, y las flechas correspondientes a los punteros.

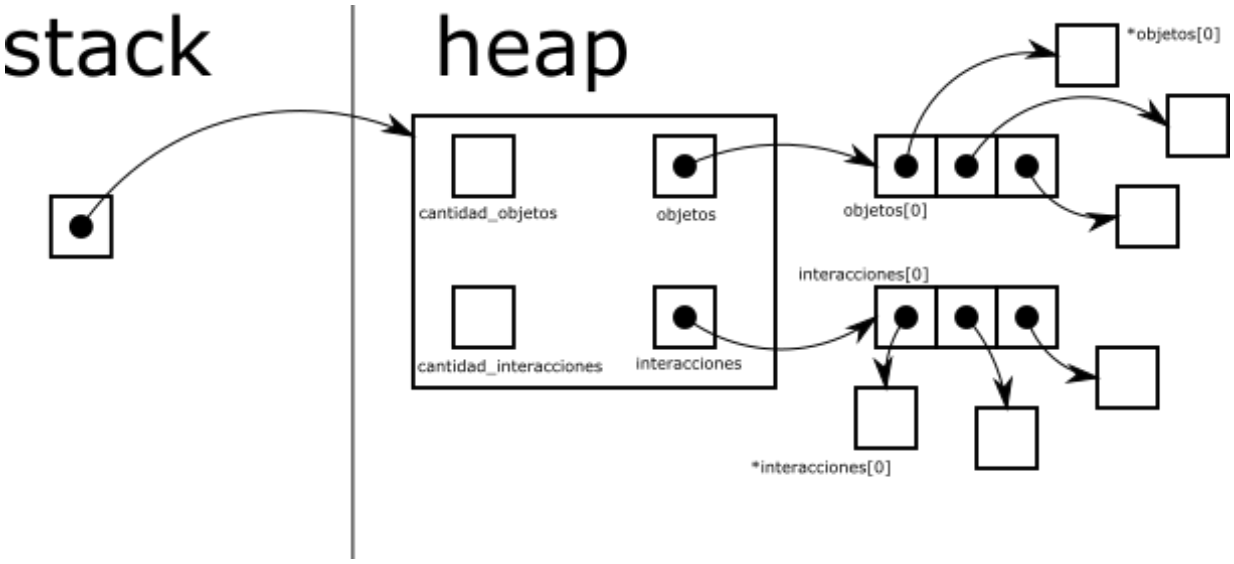


Figura 1: Memoria.