Práctica 2 - Gestión básica de listas y colas.

Programación II - Grado en Ingeniería Informática (USC)

23 de febrero de 2022

1. Objetivos

En esta práctica trataremos de:

- Reutilizar los TAD lista y cola, ya implementados y discutidos en clase de teoría.
- Valorar qué TAD es el más adecuado para el diseño de las diferentes tareas de un programa.

2. Descripción general del programa

En esta práctica utilizaremos los TAD lista y cola para implementar una simulación (parcial) del proceso de vacunación en un "vacunódromo" (centro de vacunación). En estos centros de vacunación se acumulan colas de pacientes para recibir las dosis de su cartilla de vacunación. El objetivo del programa es gestionar que los pacientes reciban las dosis que le correspondan de manera correcta, ya que una sobredosis de las mismas podría ser mortal.

El programa simulará la llegada de los pacientes al vacunódromo a la cola de vacunación. Los pacientes se gestionan por orden de llegada. Cada paciente conoce las vacunas que debe recibir, así como el número de dosis de su pauta de vacunación y el número de dosis recibidas hasta el momento. Cada paciente al frente de la cola se vacuna tan pronto como un sanitario esté disponible. El programa finaliza cuando se cierra el centro de vacunación al acabar el día, momento en el que todos los pacientes pendientes son enviados de vuelta a casa sin su vacuna.

3. Especificaciones básicas del problema

- El usuario de la aplicación inicia el programa y se le presenta un menú que permite: *(i)* añadir un nuevo paciente a la cola, *(ii)* mostrar información completa sobre el paciente de la cola al que le toca vacunarse, *(iii)* proceder a la vacunación del paciente, y *(iv)* cerrar el vacunódromo.
- Inicialmente, el vacunódromo no tiene pacientes.
- Los pacientes se almacenarán en un TAD cola (FIFO), pudiendo almacenar tantos de los mismos como sea necesario.
- Las vacunas asociadas a un paciente se almacenarán en un TAD lista.

- A la hora de crear un paciente se le preguntará, en primer lugar, su nombre. Posteriormente, se le preguntarán, una a una, cada una de las vacunas que vaya a recibir.
- Para cada vacuna se le preguntará su nombre y el número de dosis que debe recibir de la misma. El proceso de pregunta sobre las vacunas finaliza cuando el nombre de la vacuna sea "fin". El número de dosis recibidas de la vacuna es inicialmente siempre 0.
- Cuando se vacuna a un paciente, se incrementa en una unidad el número de dosis recibidas de cada una de las vacunas de su cartilla de vacunación. Si una vacuna tiene su pauta completa, se elimina de la cartilla de vacunación. Así, se distinguen dos casuísticas tras vacunar a un paciente:
  - El paciente no tiene más vacunas pendientes: se envía el paciente a casa (i.e, se elimina al paciente del programa).
  - El paciente tiene vacunas pendientes: se envía al paciente al final de la cola.
- Si se sale del programa se elimina a todos los pacientes de la cola y se elimina el vacunódromo.

4. Descripción detallada del programa.

El programa se iniciará en el menú de selección de opciones e inicializará la cola de pacientes del vacunódromo vacía. En el menú se mostrarán las siguientes opciones:

- **Nuevo paciente**: esta opción permitirá añadir un nuevo paciente al vacunódromo. Al añadir un nuevo paciente se preguntará el nombre del mismo y las vacunas que va a recibir. Para cada vacuna se preguntará su nombre y el número de dosis totales de la pauta de vacunación (el número de dosis administradas se inicializa a 0 por defecto). No se permitirá introducir vacunas con el nombre vacío o vacunas repetidas en la cartilla de vacunación. Se permitirá la introducción de pacientes con el mismo nombre (corresponderían a pacientes distintos).
- **Imprimir vacunódromo**: Se imprimirá el número de pacientes restantes en la cola de vacunación. También se imprimirá de forma detallada **únicamente** el primer paciente de la cola de vacunación. Esto implica imprimir su nombre y la información de **todas** sus vacunas. Para cada vacuna se imprimirá el nombre de la misma, junto con el máximo de dosis de la pauta de vacunación y el número de dosis administradas.
- **Vacunar siguiente paciente**: esta opción permitirá vacunar al primer paciente de la cola de vacunación. Esto implica aumentar en una unidad las dosis administradas para cada una de las vacunas de la cartilla de vacunación del paciente. Las vacunas que hayan alcanzado la pauta de vacunación completa se eliminan de la lista. Si el paciente no tiene vacunas pendientes se elimina del vacunódromo. Si el paciente tiene vacunas pendientes se reinserta al final de la cola de vacunación.
- **Salir**: esta opción elimina el vacunódromo junto con todos los pacientes y las vacunas pendientes de la cartilla de vacunación.

5. Uso del TAD

El programa deberá implementar obligatoriamente la lista y la cola indicadas con los TAD correspondientes, cuya descripción y funcionalidades se han explicado en clase de teoría.

Antes de iniciar el desarrollo de la práctica debéis repasar detalladamente las diapositivas y notas de teoría de los temas correspondientes y, de forma especial, los ejemplos, para recordar el funcionamiento de los TAD.

Para la cola de pacientes se utilizará el TAD Cola implementado en los ficheros adjuntos (`cola.c` y `cola.h`) en el que TIPOELEMENTOCOLA es una estructura con la información del paciente (nombre y lista de vacunas). Para la lista de vacunas se utilizará el TAD Lista implementado en los ficheros adjuntos (`lista.c` y `lista.h`) en el que el TIPOELEMENTOLISTA es una estructura que almacena la información de la vacuna (nombre, dosis máximas y dosis administradas).

No podrá realizarse ninguna modificación sobre los ficheros de los TAD proporcionados (ni en los .c ni en los .h) excepto la modificación del TIPOELEMENTOLISTA en `lista.h` y `lista.h` y la modificación de TIPOELEMENTOCOLA en `cola.c` y `cola.h`. También se permite la modificación de los .h para añadir los `#include` necesarios para modificar los tipos de elemento de los TAD.

6. Temporización recomendada.

A continuación se presenta una temporización recomendada para realizar cada una de las fases del proyecto. Tened en cuenta que esto que se propone aquí es sólo una estrategia de implementación: tenéis completa libertad para implementar las funcionalidades que se piden en el enunciado, siempre y cuando utilicéis el TAD cola y el TAD lista tal y como se ha especificado en los apartados anteriores.

6.1. Fase 1: Preparación del proyecto.

En primer lugar, debéis crear un proyecto en Netbeans (o en otro IDE) con el nombre "Vacunodromo" y decirle que sí queréis crear main (fijaos que a la derecha esté seleccionado C, no C++). Netbeans crea su propio makefile, os recomendamos que trabajéis con él para poder utilizar las herramientas de depuración y, una vez finalizado, construyáis vuestro makefile para realizar la entrega. Una vez que Netbeans crea la carpeta del proyecto, **debéis crear dentro de ella la subcarpeta TAD**, donde copiaréis los archivos que os pasamos en el campus virtual con la práctica y que están dentro del TAD.zip: `lista.h`, `lista.c`, `cola.h` y `cola.c`. Para poder trabajar con el terminal, tened en cuenta que Netbeans creará el ejecutable en la carpeta `./dist/Debug/Cygwin-Windows`, por lo que podéis abrir el terminal en Netbeans y cambiaros a esa carpeta para probar las entradas por línea de comandos.

6.2. Fase 2: Preparación de los TAD

6.2.1. Preparación del TAD lista (de vacunas)

En este apartado debéis preparar el `TIPOELEMENTOLISTA` como una estructura con los campos `nombreVacuna` (podéis usar un array estático, pero entonces debéis definir la longitud como una constante), y dos enteros que almacenen el número máximo de dosis y las dosis administradas.

6.2.2. Preparación del TAD cola (de pacientes)

En este apartado debéis preparar el `TIPOELEMENTOCOLA` como una estructura con los campos `nombrePaciente` (igual que antes podéis usar un array estático definiendo su longitud como una constante) y una `TLISTA` de vacunas.

6.2.3. Preparación del tipo de dato vacunódromo

En este apartado debéis definir los datos de vuestro vacunódromo, para lo que podéis definir un tipo de dato `VACUNODROMO` que sea una estructura con los campos `nombreVacunodromo` (un array de char estático o dinámico), un entero que almacene el número de pacientes en cola y una `TCOLA` de pacientes. Tanto la definición de este tipo de datos como sus funciones (que se explican en la fase 3) las podéis tener en `main.c` o en ficheros separados (por ejemplo: `vacunodromo.h` y `vacunodromo.c`).

6.3. Fase 3: Implementación del vacunódromo.

Posteriormente, deberíais implementar ya el vacunódromo. Debería haber una función `crear_vacunodromo()` que inicialice la cola de pacientes, el número de pacientes en cola, y el nombre del vacunódromo.

Tenéis que tener en cuenta que la funciones para destruir e imprimir podéis implementarlas en cadena, es decir, `destruir_vacunodromo()` llamaría a `destruir_paciente()` y `destruir_paciente()` llamaría a `destruir_vacuna()` (esta última es necesaria sólo si creáis los nombres de vacunas con memoria dinámica). El procedimiento es similar para la impresión por pantalla. También es importante recordar que tendréis que destruir (liberar memoria) individualmente para cada uno de los pacientes almacenados en la cola de pacientes, y proceder de igual manera con la lista de vacunas para cada paciente, si fuese necesario.

6.4. Fase 4: Implementación del menú.

Por último, deberéis implementar el menú del programa. Las opciones del menú que debería haber implementadas son las siguientes:

- **Imprimir vacunódromo**: tendréis que llamar a la función `imprimir()` que llame a `imprimir_paciente()`, que deberá imprimir la información del primer paciente de la cola y sus vacunas (para lo que podría llamarse a `imprimir_vacuna()`, como se explicó en la fase 3).
- **Vacunar siguiente paciente**: tendréis que llamar a la función `vacunar_paciente()` que gestionará las casuísticas de donde ubicar al paciente en función del número de vacunas en su lista de vacunación: si no tiene vacunas pendientes se elimina y si tiene vacunas pendientes se reinserta al final de la cola.
- **Añadir paciente**: tendréis que llamar a la función `anadir_paciente()` que realiza la toma de datos por teclado de la información del paciente a añadir a la cola de vacunación. Tenéis que tener en cuenta que debéis implementar un bucle para poder tomar información sobre las vacunas mientras el nombre de la misma sea distinta de "fin" (puede resultar útil aquí la función `strcmp()` incluida en la librería `string.h`).
- **Salir**: tendréis que llamar a la función `destruir_vacunodromo()` implementada en la fase 3.

7. Mejoras opcionales

7.1. Vacunas repetidas

Tal y como se ha especificado en apartados anteriores, se permite la inserción de vacunas con el mismo nombre de tal manera que la vacuna introducida con el nombre repetido se trata como una vacuna distinta. Con esta mejora, se añade un poco más de control sobre

este aspecto. Al implementar esta mejora, si se intenta insertar una vacuna con un nombre repetido, el número de dosis máximo de esta vacuna a introducir nueva (asumiendo que sea válido), se añadirá a la vacuna ya existente. Por ejemplo, si tenemos almacenado en la lista de vacunas una vacuna representada por una tupla "(nombre de la vacuna, dosis maximas)" tal que la vacuna es ("Comirnarty", 2) y se intenta insertar una vacuna ("Comirnarty", 3), la nueva vacuna de la lista de vacunas será ("Comirnarty", 5).

7.2. Lectura de datos desde un fichero

Se deberá permitir la lectura de un fichero de pacientes y vacunas y la inicialización del vacunódromo a partir de este fichero. El fichero deberá tener el siguiente formato: *PACIENTE|VACUNA_1;DOSIS_VACUNA_1;VACUNA_2;DOSIS_VACUNA_2*. Un ejemplo de fichero se muestra en el siguiente fragmento de fichero:

```
Manolo|Comirnarty;2;Chiroflu;3;Pneumovax;1
Pepe|Comirnarty;1
Juan|Chiroflu;2;Pneumovax;1
```

Se asume que el fichero es siempre correcto, para no sobrecargar vuestro proyecto con comprobaciones defensivas. El programa deberá llamarse desde línea de comandos con el siguiente formato: `./ejecutable -f NOMBRE_DEL_FICHERO`.

Para poder procesar este fichero deberéis tener en cuenta las siguientes consideraciones:

- En primer lugar, deberéis procesar los argumentos por línea de comandos. Para ello, es altamente recomendable utilizar la función `getopt()` de la librería `unistd.h`[1] puesto que soporta de forma natural los parámetros nombrados.
- En segundo lugar, deberéis leer línea a línea el fichero. Para ello podéis utilizar la función `fgets()`.
- En tercer lugar, deberéis eliminar el retorno de carro de la línea leída. Para ello, podéis utilizar la siguiente función (necesitaréis la librería `string.h`).

```
void _strip_line(char * linea){
    linea[strcspn(linea, "\r\n")] = 0;
}
```

- En cuarto lugar, deberéis procesar los campos de la línea leída. Para eso, necesitaréis dos tipos de llamadas a la función `strtok()`. En el primer tipo de llamada, partiréis la línea a partir del carácter "|". Para eso, es suficiente con hacer dos llamadas a `strtok()` tal y como sigue:

```
        char * nombre_paciente = strtok(linea, "|");
        char * lista_vacunas = strtok(NULL, "|");
```

  Después, tendréis que llamar a `strtok()` sobre el fragmento que contiene las vacunas (`lista_vacunas`) usando el carácter ";" como delimitador. Deberéis almacenar una vacuna nueva por cada par de palabras procesadas por la función. También deberéis

[1] El problema de esta librería es que solo se soporta en sistemas POSIX, pero en nuestro caso da igual porque vuestro programa deberá compilar y ejecutarse en Cygwin/Linux. **Muy probablemente esta librería no os funcione si trabajáis en Windows si usáis un compilador como MSVC (Visual Studio). En ese caso deberéis hacer el procesamiento de los argumentos a mano, tal y como se especifica en la práctica anterior.**

convertir la cadena de caracteres correspondiente al número de dosis a un entero (utilizad la función `atoi()`).

8. Entregables y calificaciones.

La implementación de las fases 1, 2 y 3 y 4 se califica sobre 9 puntos.

La implementación de la mejora opcional de vacunas repetidas permite subir la nota hasta un 9.5 mientras que la mejora opcional de lectura de ficheros permite subir la nota hasta un 11.

Deberá realizarse la entrega por el Campus Virtual, donde están especificadas las fechas de entrega particulares para cada grupo y los criterios de evaluación. Las instrucciones para generar el fichero para su entrega son las siguientes:

- Deberá subirse un único fichero comprimido con el nombre `Apellido1Apellido2_2.zip`.
- Se incluirán todos los ficheros `.c` y `.h` necesarios para compilar el proyecto junto con los TAD en su subcarpeta correspondiente y el `makefile` utilizado para compilar el proyecto. Se asumirá que los ficheros TAD necesarios están en una carpeta denominada `TAD`, que se encuentra en el mismo directorio que `main.c` y `makefile`.

Cualquier ejercicio que no compile directamente con el `makefile` entregado (sin opciones) en Linux/Cygwin será evaluado con la calificación de 0. Este criterio se mantendrá en el resto de prácticas de la asignatura.

9. Ejemplo de ejecución

A continuación se muestran ejemplos de ejecución del programa. Tened en cuenta que el formato de vuestro menú puede diferir de lo aquí expuesto: esto que se muestra es sólo un ejemplo.

9.1. Flujo normal

```
> ./ejecutable
Introduce nombre del vacunodromo: Cidade da Cultura

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 3

Nombre del paciente: Manolo
   Nombre vacuna (fin para finalizar): Comirnarty
   Numero de dosis ámximas: 3
   Nombre vacuna (fin para finalizar): Chiroflu
   Numero de dosis ámximas: 3
   Nombre vacuna (fin para finalizar): fin

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
```

```
Seleccione una opcion: 3

Nombre del paciente: Pepe
   Nombre vacuna (fin para finalizar): Comirnarty
   Numero de dosis ámximas: 1
   Nombre vacuna (fin para finalizar): fin

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 1

Vacunodromo: Cidade da Cultura
Pacientes en la cola: 2
Paciente actual: Manolo
   Lista de vacunas:
      Comirnarty: 0 de 3 dosis administradas
      Chiroflu: 0 de 3 dosis administradas

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 2

Vacunando al paciente Manolo
   Vacunando a Manolo con Comirnarty
   Vacunando a Manolo con Chiroflu

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 1

Vacunodromo: Cidade da Cultura
Pacientes en la cola: 2
Paciente actual: Pepe
   Lista de vacunas:
      Comirnarty: 0 de 1 dosis administradas

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 2

Vacunando al paciente Pepe
   Vacunando a Pepe con Comirnarty

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 1

Vacunodromo: Cidade da cultura
```

```
Pacientes en la cola: 1
Paciente actual: Manolo
   Lista de vacunas:
      Comirnarty: 1 de 3 dosis administradas
      Chiroflu: 1 de 3 dosis administradas

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 2

Vacunando al paciente Manolo
   Vacunando a Manolo con Comirnarty
   Vacunando a Manolo con Chiroflu

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 1

Vacunodromo: Cidade da Cultura
Pacientes en la cola: 1
Paciente actual: Manolo
   Lista de vacunas:
      Comirnarty: 2 de 3 dosis administradas
      Chiroflu: 2 de 3 dosis administradas

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 2

Vacunando al paciente Manolo
   Vacunando a Manolo con Comirnarty
   Vacunando a Manolo con Chiroflu

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 1

Vacunodromo: Cidade da Cultura
No hay pacientes en la cola.

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 0

>
```

9.2. Lectura del fichero por línea de comandos

```
> cat archivo_pacientes.txt
Manolo|Comirnarty;2;Chiroflu;3;Pneumovax;1
Pepe|Comirnarty;1
Juan|Chiroflu;2;Pneumovax;1

> ./ejecutable -f archivo_pacientes.txt

Introduce nombre del vacunodromo: Cidade da Cultura

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 1

Vacunodromo: Cidade da Cultura
Pacientes en la cola: 3
Paciente actual: Manolo
   Lista de vacunas:
      Comirnarty: 0 de 2 dosis administradas
      Chiroflu: 0 de 3 dosis administradas
      Pneumovax: 0 de 1 dosis administradas

Vacunodromo Cidade da Cultura
1. Imprimir vacunodromo
2. Vacunar siguiente paciente
3. Nuevo paciente
0. Salir
Seleccione una opcion: 0

>
```

a. Apéndice: recomendaciones sobre modularidad y herramientas

Es altamente recomendable concentrar funcionalidades concretas en funciones reutilizables que se utilizarán múltiples veces en distintas partes del programa. Así, por ejemplo, es recomendable aglutinar en una función la funcionalidad de aumentar el número de dosis de la vacuna en una unidad o en otra función el avance la cola de vacunación.

Todo lo que se expone en este apéndice es **totalmente opcional y no será evaluable**. Sin embargo, aplicar estas técnicas os permitirá alcanzar una mayor calidad del código y asegurar que no contiene errores. Por ejemplo, utilizar valgrind os permitirá verificar que la memoria dinámica de vuestro programa se libera correctamente y eso permitirá que no perdáis puntos cuando comprobemos que vuestro programa libera memoria correctamente.

A continuación se describen dos utilidades/recomendaciones que os permitirán simplificar la implementación de la práctica.

a.1. *Profiler*: valgrind

Se recomienda encarecidamente utilizar un *profiler* que os permita analizar el uso y accesos a la memoria del programa. De tal manera, os podréis asegurar de que toda la memoria

que se reserva se libera de forma correcta al final del programa y que los accesos a la memoria son todos legales. Existen múltiples profilers, de los cuales **valgrind** (únicamente para Linux; no funciona en Cygwin), destaca por su simplicidad.

Este profiler permite comprobar que el número de `malloc` y `free` de vuestro programa se haya realizado correctamente y que los accesos a la memoria estén dentro de los límites aceptables. A continuación se muestran algunos ejemplos de uso de `valgrind`.

a.1.1. Memoria no liberada

El siguiente programa no libera correctamente toda la memoria reservada:

```
1  #include <stdlib.h>
2
3  int main(void){
4      int * array_1 = (int*)malloc(10*sizeof(int));
5      int * array_2 = (int*)malloc(2*sizeof(int));
6      char * array_3 = (char*)malloc(3*sizeof(char));
7      float * array_4 = (float*)malloc(2 * sizeof(float));
8      free(array_1);
9      free(array_2);
10 }
```

La compilación, ejecución y depuración del programa utilizando valgrind se muestra en la siguiente figura (el *prompt* del terminal es "(base) > "):

```
1      (base) > make ejemplo_valgrind_2
2      cc     ejemplo_valgrind_2.c   -o ejemplo_valgrind_2
3      (base) > ./ejemplo_valgrind_2
4      (base) > valgrind ./ejemplo_valgrind_2
5      ==7929== Memcheck, a memory error detector
6      ==7929== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al
           .
7      ==7929== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
           info
8      ==7929== Command: ./ejemplo_valgrind_2
9      ==7929==
10     ==7929==
11     ==7929== HEAP SUMMARY:
12     ==7929==     in use at exit: 11 bytes in 2 blocks
13     ==7929== total heap usage: 4 allocs, 2 frees, 59 bytes allocated
14     ==7929==
15     ==7929== LEAK SUMMARY:
16     ==7929== definitely lost: 11 bytes in 2 blocks
17     ==7929==    indirectly lost: 0 bytes in 0 blocks
18     ==7929==      possibly lost: 0 bytes in 0 blocks
19     ==7929==    still reachable: 0 bytes in 0 blocks
20     ==7929==         suppressed: 0 bytes in 0 blocks
21     ==7929== Rerun with -leak-check=full to see details of leaked memory
22     ==7929==
23     ==7929== For lists of detected and suppressed errors, rerun with: -s
24     ==7929== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Como se puede observar en la salida anterior, en la línea: *total heap usage: 4 allocs, 2 frees, 59 bytes allocated*, se han realizado 4 mallocs en el programa y solo dos frees. Por lo tanto, 11 bytes no han sido liberados. Si reejecutamos valgrind con la opción `-leak-check=full`, podremos ver más información al respecto.

```
(base) > valgrind --leak-check=full ./ejemplo_valgrind_2
==8536== Memcheck, a memory error detector
==8536== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8536== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8536== Command: ./ejemplo_valgrind_2
==8536==
==8536==
==8536== HEAP SUMMARY:
==8536==     in use at exit: 11 bytes in 2 blocks
==8536==   total heap usage: 4 allocs, 2 frees, 59 bytes allocated
==8536==
==8536== 3 bytes in 1 blocks are definitely lost in loss record 1 of 2
==8536== at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==8536==    by 0x10919A: main (in /home/efren/Escritorio/Proyectos/programacion-ii_practica-2/Practica 2/enunciado/ejemplos_enunciado/ejemplo_valgrind_2)
==8536==
==8536== 8 bytes in 1 blocks are definitely lost in loss record 2 of 2
==8536== at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==8536==    by 0x1091A8: main (in /home/efren/Escritorio/Proyectos/programacion-ii_practica-2/Practica 2/enunciado/ejemplos_enunciado/ejemplo_valgrind_2)
==8536==
==8536== LEAK SUMMARY:
==8536==    definitely lost: 11 bytes in 2 blocks
==8536==    indirectly lost: 0 bytes in 0 blocks
==8536==      possibly lost: 0 bytes in 0 blocks
==8536==    still reachable: 0 bytes in 0 blocks
==8536==         suppressed: 0 bytes in 0 blocks
==8536==
==8536== For lists of detected and suppressed errors, rerun with: -s
==8536== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Lo interesante de este caso es que este error no se detecta ni al ejecutar el programa (no da fallo de violación del segmento) ni al pasar el depurador, mientras que solo con un profiler es posible detectar este *memory leak*.

a.1.2. Doble liberación de memoria

El siguiente programa intenta liberar memoria que ya ha sido liberada anteriormente:

```c
#include <stdlib.h>

int main(void){
    int * array = (int*) malloc(10*sizeof(int));
    free(array);
    free(array);
}
```

La salida tras compilar, ejecutar y pasar el valgrind al programa es la siguiente:

```
(base) > make ejemplo_valgrind_3
cc     ejemplo_valgrind_3.c   -o ejemplo_valgrind_3
(base) > ./ejemplo_valgrind_3
free(): double free detected in tcache 2
zsh: abort (core dumped)  ./ejemplo_valgrind_3
```

```
(base) > valgrind ./ejemplo_valgrind_3
==9476== Memcheck, a memory error detector
==9476== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9476== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==9476== Command: ./ejemplo_valgrind_3
==9476==
==9476== Invalid free() / delete / delete[] / realloc()
==9476==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgpreload_memcheck-amd64-linux.so)
==9476==    by 0x10919A: main (in /home/efren/Escritorio/Proyectos/
    programacion-ii_practica-2/Practica 2/enunciado/ejemplos_enunciado/
    ejemplo_valgrind_3)
==9476==  Address 0x4c93040 is 0 bytes inside a block of size 40 free'd
==9476==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgpreload_memcheck-amd64-linux.so)
==9476==    by 0x10918E: main (in /home/efren/Escritorio/Proyectos/
    programacion-ii_practica-2/Practica 2/enunciado/ejemplos_enunciado/
    ejemplo_valgrind_3)
==9476==  Block was alloc'd at
==9476==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgpreload_memcheck-amd64-linux.so)
==9476==    by 0x10917E: main (in /home/efren/Escritorio/Proyectos/
    programacion-ii_practica-2/Practica 2/enunciado/ejemplos_enunciado/
    ejemplo_valgrind_3)
==9476==
==9476==
==9476== HEAP SUMMARY:
==9476==     in use at exit: 0 bytes in 0 blocks
==9476== total heap usage: 1 allocs, 2 frees, 40 bytes allocated
==9476==
==9476== All heap blocks were freed -- no leaks are possible
==9476==
==9476== For lists of detected and suppressed errors, rerun with: -s
==9476== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

a.1.3. Violación del segmento

En este programa se intenta acceder a memoria no reservada:

```c
#include <stdlib.h>

int main(void){
    int* array = (int*)malloc(3 * sizeof(int));
    array[0] = 1;
    array[1] = 2;
    array[2] = 3;
    printf("Definitivamente, esto no es un buen plan.\n");
    array[3] = 4;
    printf("Estamos bien\n");
}
```

La salida de la compilación, ejecución y depuración con valgrind es la siguiente:

```
    (base) > make ejemplo_valgrind_1
    cc     ejemplo_valgrind_1.c   -o ejemplo_valgrind_1
    (base) > ./ejemplo_valgrind_1
    Definitivamente, esto no es un buen plan.
    Estamos bien
    (base) > valgrind ./ejemplo_valgrind_1
    ==10513== Memcheck, a memory error detector
```

```
8     ==10513== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
          al.
9     ==10513== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
           info
10    ==10513== Command: ./ejemplo_valgrind_1
11    ==10513==
12    Definitivamente, esto no es un buen plan.
13    ==10513== Invalid write of size 4
14    ==10513==    at 0x1091BD: main (in /home/efren/Escritorio/Proyectos/
          programacion-ii_practica-2/Practica 2/enunciado/ejemplos_enunciado/
          ejemplo_valgrind_1)
15    ==10513==  Address 0x4c9304c is 0 bytes after a block of size 12 alloc'
          d
16    ==10513==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/
          valgrind/vgpreload_memcheck-amd64-linux.so)
17    ==10513==    by 0x10917E: main (in /home/efren/Escritorio/Proyectos/
          programacion-ii_practica-2/Practica 2/enunciado/ejemplos_enunciado/
          ejemplo_valgrind_1)
18    ==10513==
19    Estamos bien
20    ==10513==
21    ==10513== HEAP SUMMARY:
22    ==10513==     in use at exit: 12 bytes in 1 blocks
23    ==10513== total heap usage: 2 allocs, 1 frees, 1,036 bytes allocated
24    ==10513==
25    ==10513== LEAK SUMMARY:
26    ==10513==    definitely lost: 12 bytes in 1 blocks
27    ==10513==    indirectly lost: 0 bytes in 0 blocks
28    ==10513==      possibly lost: 0 bytes in 0 blocks
29    ==10513==    still reachable: 0 bytes in 0 blocks
30    ==10513==         suppressed: 0 bytes in 0 blocks
31    ==10513== Rerun with --leak-check=full to see details of leaked memory
32    ==10513==
33    ==10513== For lists of detected and suppressed errors, rerun with: -s
34    ==10513== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
          0)
```

Como se puede observar del programa anterior ni la compilación ni ejecución del programa dan ningún problema[2]. Sin embargo, valgrind detecta que hay una escritura no válida (*invalid write*) en memoria no reservada. En otros casos, si el array es más grande, es posible que el programa diese un fallo de violación del segmento, lo que facilitaría la depuración del error.

a.2. *Suite* de pruebas: uso de assert.

Dado que la mayoría de la interacción del programa se realiza vía entrada y salida de teclado, la depuración del mismo puede ser especialmente tediosa. Se recomienda encarecidamente crear una *suite de pruebas* que os permitan comprobar de forma automática el funcionamiento correcto del programa. Para ello, puede seros especialmente útil la macro `assert()` (de la librería assert.h). Esta macro comprueba si la expresión que se le pasa como parámetro se evalúa a TRUE o a FALSE. Si es TRUE no hace nada, pero, si es FALSE, se muestra un error por el *stream* de error (`stderr`) y aborta el programa. Por ejemplo, dado el siguiente fragmento de código:

[2] En concreto, esto que sucede aquí se denomina "comportamiento no definido" (**undefined behaviour**). La especificación del lenguaje C no define que sucede cuando se accede a una posición de un vector que no está reservada, así que cualquier cosa puede pasar. En el mejor de los casos dará un fallo de violación del segmento y el peor de los casos el programa aparentará funcionar correctamente.

```c
#include <assert.h>
int main(void){
    assert(1 == 0);
}
```

La salida sería:

```
m: m.c:4: main: Assertion ‘1 == 0’ failed.
zsh: abort (core dumped)  ./m
```

Así, deberíais tener una función `test()` que llame a cada uno de vuestros casos de prueba (cada uno de los tests sobre vuestras funciones). Cada caso de prueba invocará a la función assert según corresponda. Un ejemplo de esqueleto de la *suite* de pruebas es el siguiente:

```c
void _test_1(){
    ...
}
void _test_2(){
    ...
}
void _test_3(){
    ...
}

void run_tests()
{
    void (*tests[])() = {
        _test_1, _test_2, _test_3
    };
    size_t n = sizeof(tests) / sizeof(tests[0]);
    for (int i = 0; i < n; i++)
    {
        (*tests[i])();
    }
}
```

```c
#define DEBUG

void main(void){
    #ifndef DEBUG
        ejecuta_menu();
    #else
        run_tests();
    #endif
}
```

En el código anterior, se crea un array de punteros a funciones que representan cada uno de los casos de prueba. Después, se itera a lo largo de ellos invocándolos uno a uno.[3] Para cambiar entre la ejecución normal del programa y los tests podéis usar una macro “DEBUG” como sigue:

Un ejemplo más completo, que integra el uso de assert, se puede observar en el siguiente fragmento de código:

[3] También sería posible llamar a las funciones una a una, pero así es más “elegante” además de permitir determinar mediante un `printf` dentro del bucle en que parte del test estamos.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

#define DEBUG

#define identifica_test() printf("Testeando funcion %s.\n", __func__)

void menu_principal(){
    printf("Este es el menu principal del programa.\n");
}

int _aprobar(char * alumno){
    printf("El alumno %s esta aprobado\n", alumno);
    return 5;
}

void _test_comprueba_alumno(){
    identifica_test();
    // Esto deberia ser un struct o llamar a una de vuestras funciones de creacion
    char* alumno = "Alumno";
    // Strcmp devuelve 0 si las cadenas son iguales
    assert(strcmp(alumno, "Alumno") == 0);
}
void _test_aprobar_alumno(){
    identifica_test();
    char* alumno = "Pepe";
    int nota_alumno = _aprobar(alumno);
    assert(nota_alumno >= 5);
}
void _test_mallocs(){
    identifica_test();
    char* alumno = (char*)malloc(4 * sizeof(char));
    assert(alumno != NULL);
    strcpy(alumno, "Pepe");
    assert(strcmp(alumno, "Pepe") == 0);
    free(alumno);
}

void run_tests()
{
    void (*tests[])() = {
        _test_comprueba_alumno,
        _test_aprobar_alumno,
        _test_mallocs
    };
    size_t n = sizeof(tests) / sizeof(tests[0]);
    for (int i = 0; i < n; i++)
    {
        printf("=======\nEjecutando test numero %d\n", i);
        (*tests[i])();
        printf("=========\n");
    }
}

int main(void){
    #ifndef DEBUG
        menu_principal();
    #else
        run_tests();
```

```
62      #endif
63  }
```

En el código anterior, la **macro** `identifica_test` os permite identificar a la función llamante utilizando el identificador `__func__` del compilador gcc. Este identificador almacena la función que está siendo llamada en cada momento. Esto no se puede convertir en una función porque, en ese caso, `__func__` devolvería el nombre de esa función llamada.

Por último, mencionar que la función `strcmp` os puede resultar útil a la hora de comparar cadenas de caracteres.