Tutorial de Valgrind y Helgrind

Introducción

Valgrind es un conjunto de herramientas libres que ayudan en la depuración de problemas de memoria y rendimiento de nuestras aplicaciones.

La herramienta en la que vamos a hacer hincapié, en un primer lugar, es **Memcheck**. Memcheck hace un seguimiento del uso de memoria de nuestro programa y puede detectar los siguientes problemas:

- Uso de memoria no inicializada (no haber hecho `malloc`).
- Lectura/escritura de memoria que ya fue liberada (con `free`).
- Lectura/escritura fuera de los límites de los bloques de memoria dinámica (escribir más de lo que pedí en el `malloc`).
- Memory leaks (no haber hecho `free`).

¿Cómo lo instalo?

En la terminal de Ubuntu, escribir:

```sh
sudo apt-get install valgrind
```

Una vez finalizada la descarga ya vamos a poder usarlo 😃.

Si estás usando una de las VMs proporcionadas por la cátedra, Valgrind ya viene instalado por defecto. Incluso en Eclipse viene instalado un plugin que nos puede ser muy útil, aunque recomendamos usar la herramienta a través de una terminal.

¿Cómo lo uso?

En la terminal, para empezar a depurar basta con escribir:

```sh
valgrind <parametros> ./<miPrograma> <argumentos>
```

Si nuestro programa no tiene ningún argumento, simplemente no hay que poner nada. Ídem si no queremos pasarle ningún parámetro a Valgrind.

Algunas opciones por parámetro copadas:

- `--leak-check=full`: habilita el detector de memory leaks.
- `--track-origins=yes`: cuando usamos memoria sin inicializar, permite rastrear en dónde se originó (lo veremos en un ejemplo más adelante).

- `--log-file=<nombreArchivo>` : crea un log de lo que nos muestra Valgrind por pantalla. Cuando tenemos muchos errores, leer de consola es un embole e incluso podemos llegar a no verlos todos, entonces guardar en un archivo toda la información que nos tira.
- `--help` : para una info detallada de todos los tipos de opciones que podemos habilitar.

> **TIP**
>
> Si ejecutamos `valgrind` sin ningún parámetro, él mismo se encargará de **aconsejarnos** parámetros importantes que nos pueden ayudar en el proceso de depuración. Por ejemplo, si no habilitamos la opción `--leak-check` y existen memory leaks en nuestro programa, Valgrind nos dejará un mensaje mágico de este tipo:
>
> ```
> Rerun with --leak-check=full to see details of leaked memory.
> ```

A continuación vamos a ver algunos ejemplos cortos y sencillos de errores de código que Memcheck detecta. La idea es dar una leve orientación a los tipos de problemas con los que nos podamos encontrar.

Invalid write of size...

c

```c
#include <stdlib.h>

int main(void) {
    char *array = malloc(5 * sizeof(char));
    array[5] = 'q';
    return 0;
}
```

Antes de empezar a hablar sobre cómo usar Valgrind, veamos qué quiero hacer:

- Crear un vector de caracteres que puede contener 5 caracteres (5 bytes en memoria).
- En la última posición del array, asignar el caracter `'q'` .
- Terminar la ejecución del programa retornando 0.

Ahora, ¿hice yo realmente lo que quería? Veamos.

Corramos nuestro programa sin usar ninguna herramienta. Tipeamos `./ej1` en consola... no tira segmentation fault, ¡todo bien entonces!.

Ok, ponele. Usemos a nuestro amigo Valgrind que es gratis.Tipeamos en consola `valgrind ./ej1` y veamos qué nos tira. Deberían ver algo similar a esto:

```
==4412== Invalid write of size 1
==4412==    at 0x40053A: main (ej1.c:5)
==4412==  Address 0x51f1045 is 0 bytes after a block of size 5 alloc'd
==4412==    at 0x4C2B3F8: malloc (in usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4412==    by 0x40052D: main (ej1.c:4)
```

> Al resto de lo que nos aparece, por el momento, ignorémoslo.

Valgrind detectó un error y nos lo está informando. Al principio realizar esta lectura es complicado, pero en unos minutos ya lo van a entender.

- ==4412== es el ID del proceso.
- La primera línea (`Invalid write...`) nos dice qué **tipo de error** es. Acá nos está diciendo que el programa escribió en una porción de memoria que no debería. `size 1` nos dice que estamos escribiendo 1 byte fuera de lo permitido.
- A continuación hay un stacktrace[^1] de la ejecución. Conviene leerlos de abajo para arriba para ir viendo dónde se origina el problema. En este caso es muy sencillo leerlo ya que tiene una sola línea: `at 0x40053A: main (ej1.c:5)` .
- Cosas como `0x40053A` son las direcciones de código, sirven para trackear bugs muuy particulares.
- En la segunda línea nos está diciendo que el problema está en la línea 5 del archivo `ej.c` (el que usé para compilar el código).
- La tercera línea informa sobre la dirección de memoria donde se está dando el problema. En este caso nos dice que la dirección de memoria `0x51f1045` que estamos escribiendo está `0 bytes after` (justo después) de un bloque de 5 bytes que fue allocado[^2] con `malloc()` en la línea 4 del mismo archivo.
- Por último, un nuevo stacktrace, contándonos en qué líneas de código se reservó el bloque de memoria en el que valgrind supone que intentamos escribir.

Los tres últimos items que mencionamos nos pueden dar una idea de por dónde puede venir el problema.

Anteriormente mencioné que yo quería, después de haber creado un vector de 5 caracteres, colocar en la última posición el caracter `'q'` . El error que estamos cometiendo es que no tuvimos en cuenta que **en C los vectores se indexan a partir del 0 en adelante**. Es decir, si `array[0]` es la primer posición del vector (el primer caracter), `array[4]` será la quinta y última posición del vector (el quinto caracter) allocada.

Como te habrás dado cuenta, `array[5]` es la sexta posición del vector, pero nosotros nunca la allocamos. Como consecuencia, estamos escribiendo en una porción de memoria que no nos pertenece (de ahí el mensaje `invalid write...`).

Uno dirá: *"ajam cerebrito, pero mi programa funcionó igual"*.

Sí, funcionó, pero cuando queramos hacer algún uso más de ese vector probablemente en algún momento explote todo. Recordemos que nosotros no controlamos los segmentos de memoria que son asignados a nuestro programa, y éstos pueden variar de ejecución en ejecución.

Entonces, que corriéndolo una vez funcione no significa que siempre vaya a funcionar. En otra ejecución o en otra máquina, el sistema operativo podría asignarnos la memoria de forma distinta, y darse la casualidad de que la posición de memoria siguiente a nuestro array de 5 caracteres no nos pertenezca, por lo que en ese momento sí nos va a dar un `segmentation fault` .

Recordemos también que siempre que pueda fallar algo en el trabajo ~~práctico~~, fallará en producción ~~o en la entrega~~. No queremos eso. El chiste de correr valgrind es que, al garantizar que no hay

problemas de memoria, garantizamos que el programa no va a fallar con segmentation faults nunca[^3].

Estos errores son comunes, sobre todo cuando recién empezamos a programar con manejo de memoria dinámica, o porque estamos despistados, o porque hace días venimos desarrollando para llegar a la entrega. Valgrind nos da una mano detectando estos errores que nos pueden dar dolores de cabeza por horas.

Entonces, hagamos el cambio:

```c
#include <stdlib.h>

int main(void) {
    char *array = malloc(5 * sizeof(char));
    array[4] = 'q';
    return 0;
}
```

Volvemos a correr Memcheck, ¡desapareció el error! Sin embargo...

```
==6789== HEAP SUMMARY:
==6789==     in use at exit: 5 bytes in 1 blocks
==6789==   total heap usage: 1 allocs, 0 frees, 5 bytes allocated
==6789==
==6789== LEAK SUMMARY:
==6789==    definitely lost: 5 bytes in 1 blocks
==6789==    indirectly lost: 0 bytes in 0 blocks
==6789==      possibly lost: 0 bytes in 0 blocks
==6789==    still reachable: 0 bytes in 0 blocks
==6789==         suppressed: 0 bytes in 0 blocks
==6789== Rerun with --leak-check=full to see details of leaked memory
```

- En la tercera línea nos está diciendo que allocamos memoria una vez, pero nunca la liberamos (nunca llamamos al `free()`).
- En la última línea se observa la magia de Valgrind. Nos está avisando que hay memory leaks y nos aconseja volverlo a correr con el parámetro `--leak-check=full` para obtener más detalles.

> **TIP**
>
> Si estás pensando *"¿Cómo? ¿En la columna de la izquierda no decía `==4412==` ?"*, recordá que el ID es del proceso (o sea, "programa en ejecución"), y no del programa. Por eso cambia en cada ejecución, y tus ID seguramente sean distintos a los que ves acá.

Definitely lost

Habiendo solucionado el error en la asignación, vamos correr nuevamente el código del ejemplo anterior, con la diferencia de que le pediremos a Valgrind que haga un chequeo de memory leaks.

```c
#include <stdlib.h>

int main(void) {
    char *array = malloc(5 * sizeof(char));
    array[4] = 'q';
    return 0;
}
```

Tipeamos en consola para detectar los memory leaks:

```sh
valgrind --leak-check=yes ./ej1
```

Deberían ver algo similar a esto:

```
==5263== HEAP SUMMARY:
==5263==     in use at exit: 5 bytes in 1 blocks
==5263==   total heap usage: 1 allocs, 0 frees, 5 bytes allocated
==5263==
==5263== LEAK SUMMARY:
==5263==    definitely lost: 5 bytes in 1 blocks
==5263==    indirectly lost: 0 bytes in 0 blocks
==5263==      possibly lost: 0 bytes in 0 blocks
==5263==    still reachable: 0 bytes in 0 blocks
==5263==         suppressed: 0 bytes in 0 blocks
==5263==
==5263== 5 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5263==    at 0x4C2B3F8: malloc (in usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5263==    by 0x40052D: main (ej1.c:4)
```

- Como vimos en la **guía de punteros y memoria dinámica**, el HEAP es un área grande de memoria a donde recurrimos al momento de reservar memoria dinámicamente con `malloc()` .
- En la tercera línea, nos dice que allocamos una vez (`1 allocs`), no liberamos nunca (`0 frees`) y que son 5 los bytes allocados.
- En la sexta y la doceava línea nos dice que perdimos **definitivamente** 5 bytes en un bloque de memoria. Claramente tenemos que arreglarlo.

> **Una situación común en la que esto nos puede perjudicar seriamente**
>
> Tal vez para llevar a cabo ciertas tareas tengamos que llamar a una función específica muy a menudo. Supongamos que esta función realiza una operación X haciendo uso de unas estructuras auxiliares que son allocadas en memoria dinámica. Estas estructuras auxiliares nos dejarán de ser útiles al momento en que se termine de ejecutar el bloque de código de la función.
>
> Como están allocadas en memoria dinámica, nosotros somos responsables de liberar las porciones que pedimos. Si nunca las liberamos, en un momento nuestro programa va a haber consumido una gran cantidad de memoria y esto va a influir en la performance (incluso **podemos quedarnos sin memoria disponible**).

Para evitar esto, agregaremos el `free()` correspondiente:

```c
#include <stdlib.h>

int main(void){
    char *array = malloc(5 * sizeof(char));
    array[4] = 'q';
    free(array);
    return 0;
}
```

Si volvemos a correr valgrind, veremos que nos dirá:

```
All heap blocks were freed -- no leaks are possible
```

En otras palabras, no tenemos más leaks.

Conditional jump or move depends on uninitialised values

En el stack

```c
#include <stdio.h>

int main(void) {
    int a;
    printf("a = %d \n", a);
    return 0;
}
```

¿Qué hace nuestro programa?

- Declara una variable entera (`int`) de nombre `a` .
- Imprime en pantalla el valor de la variable.
- Termina la ejecución del programa retornando 0.

Tipeamos en consola `./ej3` y nos muestra en pantalla lo siguiente:

```
a = 0
```

Ahora tipeamos en consola `valgrind ./ej3` y nos muestra el siguiente mensaje:

```
==7079== Conditional jump or move depends on uninitialised value(s)
==7079==    at 0x4E7C4F1: vfprintf (vfprintf.c:1629)
==7079==    by 0x4E858D8: printf (printf.c:35)
==7079==    by 0x400537: main (ej3.c:5)
==7079==
==7079== Use of uninitialised value of size 4
==7079==    at 0x4E7A7EB: _itoa_word (_itoa.c:195)
==7079==    by 0x4E7C837: vfprintf (vfprintf.c:1629)
```

```
==7079==    by 0x4E858D8: printf (printf.c:35)
==7079==    by 0x400537: main (ej3.c:5)
```

> **Antes de analizar el mensaje...**
>
> Es importante entender que la porción de memoria que se le asigna a una variable podría contener basura.
>
> El kernel de Linux se encarga de llenar la memoria con ceros[^4] (por lo que nuestras variables no inicializadas siempre tendrian 0) pero POSIX en sí no garantiza nada. Siempre conviene respetar el contrato en lugar de los detalles de implementación.
>
> También tengamos en cuenta que, como programadores, declarar una variable y nunca darle un valor inicial antes de empezar a trabajar con ella es una mala práctica.

- En la primera línea vemos que el tipo de error es `Conditional jump or move depends on uninitialised value(s)`, es decir, se evalúa un salto condicional con un valor no inicializado.
- Leemos el stack trace de abajo para arriba. Comenzamos en la función `main()`, en la línea 5 de nuestro código (`main (ej3.c:5)`) donde llamamos a `printf()`. Internamente, `printf()`, en la línea 35 (`printf (printf.c:35)`), llama a la función `vfprintf()`. vfprintf, en la línea 1629, hace una llamada y... Memcheck putea.
- Y si lo anterior no bastó, pasemos al mensaje de la línea 6: `Use of uninitialised value of size 4`. Más claro imposible, Valgrind nos está diciendo que estamos haciendo uso de un valor de tamaño de 4 bytes que no inicializamos.

> **Para tener en cuenta...**
>
> Memcheck nos deja utilizar las variables no inicializadas hasta cierto punto.
>
> De hecho, si quitamos la línea del printf y volvemos a correr Valgrind, nos va a decir que está todo bien. ¿Por qué? Porque **Memcheck solo va a putear cuando los datos no inicializados afecten el comportamiento externo/visible de nuestro programa**.
>
> En este caso particular, Memcheck putea en el vfprintf porque internamente tiene que *examinar* el valor contenido en la variable `a` para convertirlo en la cadena que corresponda en ASCII (sí, es acá donde ocurre el salto condicional).

Entonces, sólo nos queda darle un valor inicial a nuestra variable para que Valgrind no se enoje.

```c
#include <stdio.h>

int main(void) {
    int a = 1;
    printf("a = %d \n", a);
    return 0;
}
```

Volvemos a correr nuestro programa y... ¡no hay errores!. Ahora imprime a = 1 y tanto nosotros como Valgrind somos felices.

¿Cómo se daría este problema en memoria dinámica? 🤔

En memoria dinámica

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

int main(void) {
    int *a = malloc(sizeof(int));
    printf("a = %d \n", (*a));
    free(a);
    return 0;
}

```

¿Qué hace nuestro programa?

- Declara un puntero de nombre `a`, al cual le allocamos en memoria dinámica un espacio equivalente al tamaño de una variable tipo entero.
- Imprime en pantalla el valor de la variable.
- Libera el puntero.
- Termina la ejecución del programa retornando 0.

> **Para tener en cuenta...**
>
> Esta es la diferencia entre los dos operadores básicos para trabajar con punteros:
>
> `*` me permite acceder al **contenido** de la dirección de memoria (dato) a la que apunta el puntero.
>
> `&` me permite saber **la dirección en memoria** de la variable.
>
> Como en el printf quiero acceder al valor al que apunta el puntero, debo usar el primer operador: (`*a`).

Si hacemos un `valgrind ./ej4` en consola vamos a ver que los mensajes de error son similares a los del código anterior.

```
==13230== Conditional jump or move depends on uninitialised value(s)
==13230==    at 0x4E7C4F1: vfprintf (vfprintf.c:1629)
==13230==    by 0x4E858D8: printf (printf.c:35)
==13230==    by 0x4005D8: main (ej3.c:6)
==13230==
==13230== Use of uninitialised value of size 8
==13230==    at 0x4E7A7EB: _itoa_word (_itoa.c:195)
==13230==    by 0x4E7C837: vfprintf (vfprintf.c:1629)
==13230==    by 0x4E858D8: printf (printf.c:35)
==13230==    by 0x4005D8: main (ej3.c:6)
==13230==
==13230== Conditional jump or move depends on uninitialised value(s)
==13230==    at 0x4E7A7F5: _itoa_word (_itoa.c:195)
==13230==    by 0x4E7C837: vfprintf (vfprintf.c:1629)
```

```
==13230==    by 0x4E858D8: printf (printf.c:35)
==13230==    by 0x4005D8: main (ej3.c:6)
```

La solución, nuevamente, es inicializar nuestra variable.

Como estamos trabajando con punteros, debemos tener en cuenta que a lo que le vamos a asignar un valor va a ser al contenido de la dirección a la que apunta el puntero, por ejemplo: `*a = 1;`

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

int main(void) {
    int *a = malloc(sizeof(int));
    (*a) = 1;
    printf("a = %d \n", (*a));
    free(a);
    return 0;
}
```

Usando `memcpy()`

Por último, un caso particular que nos puede ocurrir a la hora de manejar variables sin inicializar es el siguiente:

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

int main(void) {
    int *a = malloc(sizeof(int));
    int *b = malloc(sizeof(int));
    memcpy(b, a, sizeof(int));
    printf("b = %d \n", (*b));
    free(b);
    return 0;
}
```

¿Qué hace nuestro programa?

- Declara dos punteros de nombre `a` y `b`, a los cuales les allocamos en memoria dinámica un espacio equivalente al tamaño de una variable tipo `int`.
- Copia la cantidad de bytes equivalente al tamaño de una variable tipo `int`, partiendo desde la dirección `a` y guardando todo en la dirección `b`.
- Imprime en pantalla el valor de `b`.
- Libera la memoria apuntada por `b`.
- Termina la ejecución del programa retornando 0.

Uno asumiría que al hacer `valgrind ./ej5` nos va a aparecer una advertencia en la línea donde se encuentra nuestro `memcpy()`, ya que se lee el contenido en la dirección de memoria `a` y se guarda en `b`.

Sin embargo, la realidad supera a la ficción...

```
==24112== Memcheck, a memory error detector
==24112== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==24112== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==24112== Command: ./ej5
==24112==
==24112== Conditional jump or move depends on uninitialised value(s)
==24112==    at 0x48DDCA6: __vfprintf_internal (vfprintf-internal.c:1646)
==24112==    by 0x48C858E: printf (printf.c:33)
==24112==    by 0x1091F6: main (ej5.c:9)
==24517==
==24517== Use of uninitialised value of size 8
==24517==    at 0x48C216B: _itoa_word (_itoa.c:179)
==24517==    by 0x48DD964: __vfprintf_internal (vfprintf-internal.c:1646)
==24517==    by 0x48C858E: printf (printf.c:33)
==24517==    by 0x1091F6: main (ej5.c:9)
==24517==
```

Valgrind chilla recién cuando se lee `b`, ¡y no nos muestra de dónde se origina esa memoria sin inicializar! 😠

Para este caso sencillo, ya podemos deducir su origen mirando el código, pero supongamos que en el TP estamos probando la serialización de un mensaje complejo con varios `memcpy()`: el hecho de que *solo* nos aparezca el error al enviar el mensaje completo (sin advertir de dónde proviene) nos puede resultar *poco descriptivo*.

Por suerte, Valgrind es sabio, y al final de los logs nos recomienda agregar una opción por parámetro para solucionar esto:

```
==24517== All heap blocks were freed -- no leaks are possible
==24517==
==24517== Use --track-origins=yes to see where uninitialised values come from
==24517== For lists of detected and suppressed errors, rerun with: -s
==24517== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

Efectivamente, la opción `--track-origins=yes` habilita el seguimiento de toda la memoria no inicializada. Volvamos a ejecutar el programa con:

sh

```
valgrind --track-origins=yes ./ej5
```

Ahora sí, al final del stack trace nos aparecerá el *verdadero* origen del warning:

```
==30024== Memcheck, a memory error detector
==30024== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==30024== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==30024== Command: ./ej5
==30024==
==30024== Conditional jump or move depends on uninitialised value(s)
==30024==    at 0x48DDCA6: __vfprintf_internal (vfprintf-internal.c:1646)
==30024==    by 0x48C858E: printf (printf.c:33)
==30024==    by 0x1091D8: main (ej5.c:9)
```

```
==30024==  Uninitialised value was created by a heap allocation
==30024==    at 0x4843839: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==30024==    by 0x10919E: main (ej5.c:6)
==30024==
==30024== Use of uninitialised value of size 8
==30024==    at 0x48C216B: _itoa_word (_itoa.c:179)
==30024==    by 0x48DD964: __vfprintf_internal (vfprintf-internal.c:1646)
==30024==    by 0x48C858E: printf (printf.c:33)
==30024==    by 0x1091D8: main (ej5.c:9)
==30024==  Uninitialised value was created by a heap allocation
==30024==    at 0x4843839: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==30024==    by 0x10919E: main (ej5.c:6)
```

En este caso, se trata de una alocación en el heap hecha por un `malloc()` en la línea 6 de nuestro archivo `ej5.c` .

De nuevo, veremos que la solución es inicializar el contenido apuntado por `a` .

Syscall param contains uninitialised bytes

```c
#include <stdlib.h>

int main(void) {
    int a;
    exit(a);
}

```

Si bien cae de maduro cuál es el error, tipeamos en consola `valgrind ./ej6` y deberíamos ver algo como esto:

```
==5758== Syscall param exit_group(status) contains uninitialised byte(s)
==5758==    at 0x4EF1C18: _Exit (_exit.c:33)
==5758==    by 0x4E6D95F: __run_exit_handlers (exit.c:93)
==5758==    by 0x4E6D984: exit (exit.c:100)
==5758==    by 0x40052D: main (ej3.c:5)
```

Valgrind detecta si al hacer una llamada al sistema[^5] (en este caso, `exit()`) estamos pasándole por parámetro variables no inicializadas.

Esto puede ser muy útil porque, teniendo en cuenta las cuestiones del estado de la memoria al momento de ejecutar nuestro proceso, nuestro programa no siempre funcionará como esperamos y no vamos a recibir ningún mensaje que nos advierta que la syscall está recibiendo un valor no inicializado[^6].

Cierre

Para recurrir a información más detallada referente a los mensajes de error y al funcionamiento de Memcheck pueden recurrir al **manual de usuario de valgrind**.

Yo no quiero venderte Valgrind ni nada por el estilo, pero la verdad es que es una herramienta muy poderosa que cualquier programador en C/C++ debería conocer. ¿Por qué?, ¡por todo lo que se estuvo hablando hasta ahora!, tus programas van a ser más eficientes en el manejo de la memoria y te va a ahorrar mucho tiempo cuando surjan segmentation faults[^7] (¡incluso te ayuda a prevenirlos!).

Otra herramienta muy útil que provee Valgrind, es **Helgrind**, para solucionar y detectar problemas de sincronización. También te va a resultar muy útil. Si no viste la charla del principio, te recomiendo al menos saltarte a la parte en la que se menciona esta herramienta.

Material recomendado

- **C Dynamic Memory Debugging with Valgrind**

[^1]: Stacktrace: reporte del estado del stack de ejecución del programa [^2]: No está loco, está *reservado* [^3]: Ni. Recordar que **los test aseguran la presencia de errores, no su ausencia**, pero bueno, me explico.

[^4]: Para más información sobre esto, leer **este artículo**, en especial el párrafo de **Uninitialized Data Segment**.

[^5]: **Acá** pueden encontrar una guía de referencia sobre las syscalls del kernel de Linux.

[^6]: Bueno, esto no es tan así, si al momento de compilar tenemos habilitadas las warnings el compilador nos advertirá que la función recibe una variable no inicializada.

[^7]: Rara vez tuve que recurrir al debugger de Eclipse para solucionar este tipo de problemas luego de haber usado Valgrind.