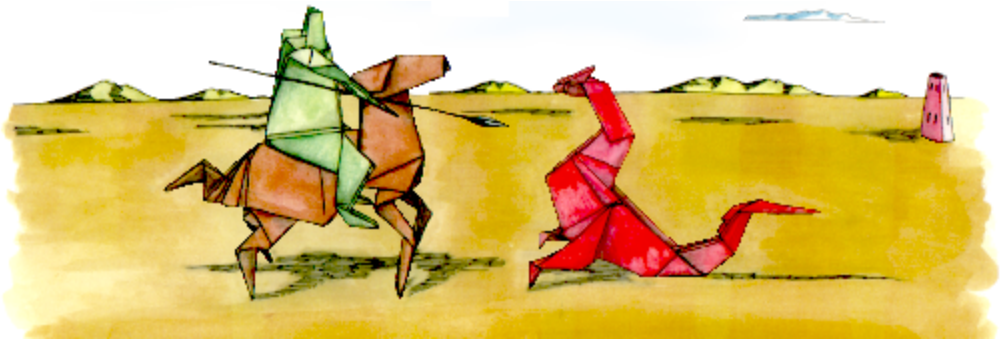
Valgrind

http://faq.utnso.com/**valgrind**



Introducción

- ¿Qué es Valgrind?
- ¿Cómo lo instalo?
- ¿Cómo lo uso?

Ejemplost

- Ejemplo 1: “Invalid write of size…”
- Ejemplo 2: Free
- Ejemplo 3: “Conditional jump or move depends on uninitialised values”
- Ejemplo 4: “Conditional jump or move depends on uninitialised values” (Ejemplo 3 en Memoria Dinámica)
- Ejemplo 5: “Syscall param contains uninitialised bytes”

Cierre

Introducció

¿Qué es Valgrind?

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 éste caso, 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 (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 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, pero la integración no es muy buena y puede confundir (bastante) a la hora de tener que usarlo desde el IDE. Por ésto último, recomendamos usar la herramienta a través de una terminal.

¿Cómo lo uso?

En la terminal, basta con escribir valgrind <parámetros> <miPrograma> <argumentos> para empezar a depurar. Si nuestro programa no tiene ningún argumento, simplemente no hay que poner nada. Idem si no queremos pasarle ningún parámetro a Valgrind.

Algunas opciones por parámetro[1] copadfree(paginaM);as :

- --leak-check=yes habilita el detector de memory leaks.
- --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

[1] 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 éste tipo: *Rerun with --leak-check=full to see details of leaked memory*.

nos va a permitir manejarnos mejor.

- --help para una info detallada de todos los tipos de opciones que podemos habilitar.

Ejemplos

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.

Ejemplo 1: “Invalid write of size...”

```c
#include <stdlib.h>

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

ej1.c

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 ésto:

```
==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 les aparece, por el momento, ignórenlo.

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

- ==4412== es el ID del proceso.
- La primer 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[2] de la ejecución. Conviene leerlos de abajo para arriba para ir viendo dónde se origina el problema. En éste caso es muy sencillo leerlo (tiene una sola línea: *at 0x40053A: main (ej1.c:5)*).
- Cosas como *0x40053A* son las direcciones de código, sirven para trackear bugs muuy raros.
- En la segunda línea nos está diciendo que el problema está en la línea 5 de nuestro código (*array[5]='q';*).
- La tercera línea informa sobre la dirección de memoria donde se está dando el problema. En éste caso nos dice que la dirección de memoria (*0x51f1045*) que estamos escribiendo está justo (*0 bytes after*) después de un bloque de 5 bytes allocado[3] con malloc() en la línea 4 de nuestro código.
- 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 estos pueden variar de ejecución en ejecución. Entonces, que corriendolo 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[4].

---

[2] Stacktrace: reporte del estado del stack de ejecución del programa
[3] No está loco, está *reservado*
[4] Ni. Recordar que *los test aseguran la presencia de errores, y no la ausencia*, pero meh.

Éstos errores son comunes, sobre todo cuando recién empezamos a programar con manejo de memoria dinámica o porque estamos despistados o porque hace días venimos sin dormir 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[5]='q';
        return 0;
}
```

Código antes de Valgrind

```c
#include <stdlib.h>

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

Código después de Valgrind

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

```
==6789[5]== 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 tercer línea (*total heap usage: 1 allocs, 0 frees, 5 bytes allocated*) nos está diciendo que allocamos memoria una vez, pero nunca la liberamos (nunca llamamos al **free()**).
- En la última línea (*Rerun with --leak-check=full to see details of leaked memory*) 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.

Ejemplo 2: Free

Habiendo solucionado el error en la asignación, vamos correr nuevamente el código del

[5] Si estás pensando "¿Cómo? ¿No era 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

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;
}
```

ej1.c

Tipeamos en consola valgrind --leak-check=yes ./ej1 para detectar los memory leaks. Deberían ver algo similar a ésto:

```
==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== 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)
```

- *Heap* es un área grande de memoria libre sin usar. En otra palabra, a donde recurrimos al momento de reservar memoria dinámica.
- En la tercer línea, nos dice que allocamos una vez (*1 allocs*), no liberamos nunca (*0 frees)* y que son 5 los bytes allocados.
- En la quinta línea nos dice que *perdimos **definitivamente** 5 bytes en un bloque de memoria*. Claramente tenemos que arreglarlo.

En otras palabras, Valgrind nos está diciendo que **nunca le hicimos un free a la porción de memoria dinámica que reservamos.**

Pensemos una situación común en la que ésto nos puede perjudicar seriamente. Tal vez para llevar a cabo ciertas tareas tengamos que llamar a una función específica muy a menudo. Supongamos que ésta función realiza una operación X haciendo uso de unas estructuras auxiliares que son allocadas en memoria dinámica. Éstas 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 ésto va a influir en la performance e incluso podemos

quedarnos sin memoria disponible.

```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.

Ejemplo 3: “Conditional jump or move depends on uninitialised values”

```c
#include <stdio.h>
int main(void){
        int a;
        printf("a = %d \n", a);
        return 0;
}
```

ej3.c

¿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 8[6]
==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)
==7079==
==7079== Conditional jump or move depends on uninitialised value(s)
==7079==    at 0x4E7A7F5: _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

[6] Puede ser que aparezca size 4, depende de la arquitectura utilizada. En la siguiente página está explicado.

asigna a una variable podría contener basura.
El kernel de Linux se encarga de llenar la memoria con ceros[7] (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 primer 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.
- 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 sólo va a putear cuando los datos no inicializados afecten el comportamiento externo/visible[8] de nuestro programa.** En éste 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).
- Y si lo anterior no bastó, pasemos al mensaje de la línea 6: "*Use of uninitialised value of size 8*"[9]. Más claro imposible, Valgrind nos está diciendo que estamos haciendo uso de un valor de tamaño de 8 bytes que no inicializamos.

[7] Para más información sobre ésto, leer éste artículo, en especial el párrafo de **Uninitialized Data Segment**.
[8] Ojo: no confundir vfprintf y "comportamiento externo/visible" con "mostrar por pantalla". Por ejemplo, utilizar una variable no inicializada en un if o en una llamada al sistema provocará el mismo tipo de error.
[9] Por definición, **int** representa tantos bytes como esté definida la palabra del procesador. Si estamos usando un Linux de 32 bits aparecerá "size 4". En mi caso particular estoy usando uno de 64 bits, por lo que aparecerá "size 8". Ésto es porque el tipo **int** varía su tamaño (y por ende, los valores que puede tomar) dependiendo de la arquitectura. Para que el rango no dependa de la arquitectura bajo la que estemos corriendo nuestro programa, podemos hacer uso de tipos como **int32_t** (int de 32 bits), **int64_t** (int de 64 bits).

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

Código antes de Valgrind

```c
#include <stdio.h>
int main(void){
        int a = 1;
        printf("a = %d \n", a);
        return 0;
}
```

Código después de Valgrind

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

¿Cómo se daría éste problema en memoria dinámica?.

Ejemplo 4: “Conditional jump or move depends on uninitialised values” (Ejemplo 3 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;
}
```

ej4.c

¿Qué hace nuestro programa?

- Declara un puntero de nombre “a” al cual le allocamos un espacio, en memoria dinámica, equivalente al tamaño de una variable tipo entero.
- Imprime en pantalla el valor de la variable.[10]
- Libera el puntero.
- Termina la ejecución del programa retornando 0.

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)
```

[10]Hay que tener en cuenta 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).

```
==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));
        printf("a = %d \n", (*a));
        free(a);
        return 0;
}
```

Código antes de Valgrind

```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;
}
```

Código después de Valgrind

Ejemplo 5: “Syscall param contains uninitialised bytes”

```c
#include <stdlib.h>
int main(void){
        int a;
        exit(a);
}
```

ej5.c

Si bien cae de maduro cuál es el error, tipeamos en consola valgrind ./ej5.c y deberíamos ver algo como ésto:

```
==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[11] (en éste caso **exit(int status)**) estamos pasándole por parámetro variables no inicializadas. Ésto 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[12].

[11] Acá pueden encontrar una guía de referencia sobre las syscalls del kernel v2.6 de Linux
[12] Bueno, ésto 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.

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 o 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[13] (¡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, pero eso ya es otra historia...

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