# 75.40 Algoritmos y Programación II Curso 4
# Valgrind

Dr. Mariano Méndez, Lucas Bonfil [1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

10 de marzo de 2020

## 1. Introduccion

A la hora de trabajar con memoria dinámica en C, Hay que tomar unas consideraciones. Mi programa pierde memoria?, ¿Entro a partes de memoria invalida?,¿Cuanta memoria pierdo?, ¿Dónde se pidió esa memoria? Todas estas preguntas son resueltas por Valgrind.

## 2. Instalación y ejecución

Antes que nada, tenemos que instalar valgrind, para eso simplemente se debe ejecutar una linea en la terminal.

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

Luego para correr valgrind se debe compilar el código, y se debe correr la siguiente linea de comandos en la terminal

```
valgrind ./*ejecutable*
```

donde se debe cambiar *ejecutable* por el nombre del programa que resultó de la compilación. Ahora, nosotros recomendamos utilizar otra linea de comandos, la cual resulta darnos información extra en caso de perdida de memoria

```
valgrind --leak-check=full --track-origins=yes --show-reachable=yes ./*ejecutable*
```

al agregar esos parámetros extra nos va a dar información extra sobre los lugares donde se pide memoria sin borrar y cuanto se pide.

A continuación Vamos a mostrar algunos ejemplos de los mensajes que puede llegar a producir valgrind.

## 3. Mensaje Esperado

En caso que el código no pierda memoria el mensaje esperado va a parecerse al siguiente.

```
==2890== HEAP SUMMARY:
==2890==     in use at exit: 0 bytes in 0 blocks
==2890==   total heap usage: ### allocs, ### frees, ### bytes allocated
==2890==
==2890== All heap blocks were freed -- no leaks are possible
==2890==
==2890== For counts of detected and suppressed errors, rerun with: -v
==2890== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

se debe notar que en lugar de ### van a ir valores numéricos, que van a depender de cada programa que corramos, y cuanta memoria se pida en el mismo.

# 4. Ejemplos sobre memoria no liberada

## 4.1. Ejemplo 1

En el siguiente ejemplo es muy trivial, en el cual vamos a pedir memoria en el main.

```c
#include<stdlib.h>
#include<stdio.h>
int main(){
  //pido espacio de memoria para un numero
  int* numero=malloc(sizeof(int*));
  //realizo una operacion cualquiera, en este caso asigno un valor
  *numero=8;
  return 0;
}
```

este ejemplo tas ejecutar el programa con valgrind nos va generar la siguiente salida:

```
==3380== Memcheck, a memory error detector
==3380== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3380== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3380== Command: ./pruebas
==3380==
==3380==
==3380== HEAP SUMMARY:
==3380==     in use at exit: 8 bytes in 1 blocks
==3380==   total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==3380==
==3380== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3380==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3380==    by 0x400537: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3380==
==3380== LEAK SUMMARY:
==3380==    definitely lost: 8 bytes in 1 blocks
==3380==    indirectly lost: 0 bytes in 0 blocks
==3380==      possibly lost: 0 bytes in 0 blocks
==3380==    still reachable: 0 bytes in 0 blocks
==3380==         suppressed: 0 bytes in 0 blocks
==3380==
==3380== For counts of detected and suppressed errors, rerun with: -v
==3380== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Si viéramos el heap sumary, podemos notar que la memoria se pidió en el main, por la función malloc. Hay que notar que valgrind no nos va a decir en que lugar hay que liberar la memoria, ni cuando fue la ultima vez que se tenia referencia a ese bloque de memoria, solo nos informa que función reservó el espacio en memoria y en que contexto.

## 4.2. Ejemplo 2

En este ejemplo vamos a reservar memoria para un array de int con un tamaño de 25.

```c
#include<stdlib.h>
#include<stdio.h>
int main(){
  //pido espacio de memoria para 25 numeros
  int* numero=malloc(sizeof(int*)*25);
  //realizo una operacion cualquiera, en este caso asigno un valor
  for (int i=0;i<25;i++){
        numero[i]=i;
  }
  return 0;
}
```

la salida producida es:

```
==3491== Memcheck, a memory error detector
==3491== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3491== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3491== Command: ./pruebas
==3491==
==3491==
==3491== HEAP SUMMARY:
==3491==     in use at exit: 200 bytes in 1 blocks
==3491==   total heap usage: 1 allocs, 0 frees, 200 bytes allocated
==3491==
==3491== 200 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3491==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3491==    by 0x400537: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3491==
==3491== LEAK SUMMARY:
==3491==    definitely lost: 200 bytes in 1 blocks
==3491==    indirectly lost: 0 bytes in 0 blocks
==3491==      possibly lost: 0 bytes in 0 blocks
==3491==    still reachable: 0 bytes in 0 blocks
==3491==         suppressed: 0 bytes in 0 blocks
==3491==
==3491== For counts of detected and suppressed errors, rerun with: -v
==3491== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Se puede notar que en heap summary el se obtiene que se perdió un bloque de 200 bytes, lo cual nos puede dar una pista bastante interesante para saber que es lo que no se libera.

## 4.3. Ejemplo 3

[language=c] En el siguiente ejemplo, vamos a pedir memoria a través de funciones

```c
#include<stdlib.h>
#include<stdio.h>
int* pedir_memoria(){
        int* numero=malloc(sizeof(int));
        return numero;
}

void funcion_que_pide_memoria_dos_veces(){
        pedir_memoria();
        pedir_memoria();
}

int main(){
  int* numero=pedir_memoria();
  funcion_que_pide_memoria_dos_veces();
  *numero=8;
  return 0;
}
```

Podemos notar que la función pedir_memoria() reserva un espacio de memoria dinamica, por otro lado si vemos a funcion_que_pide_memoria_dos_veces(), vemos que llama dos veces a pedir_memoria(), pero que no hace absolutamente nada con esa memoria reservada. Veamos cual es el mensaje producido por valgrind.

```
==3571== Memcheck, a memory error detector
==3571== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3571== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3571== Command: ./pruebas
==3571==
==3571==
==3571== HEAP SUMMARY:
==3571==     in use at exit: 12 bytes in 3 blocks
==3571==   total heap usage: 3 allocs, 0 frees, 12 bytes allocated
==3571==
==3571== 4 bytes in 1 blocks are definitely lost in loss record 1 of 3
==3571==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3571==    by 0x400537: pedir_memoria (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3571==    by 0x40056E: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3571==
==3571== 4 bytes in 1 blocks are definitely lost in loss record 2 of 3
==3571==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3571==    by 0x400537: pedir_memoria (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3571==    by 0x40054F: funcion_que_pide_memoria_dos_veces (in
/home/lucas/Escritorio/2019 Mendez algo2/ejemplos_valgrind/pruebas)
==3571==    by 0x40057C: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3571==
==3571== 4 bytes in 1 blocks are definitely lost in loss record 3 of 3
==3571==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3571==    by 0x400537: pedir_memoria (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3571==    by 0x400559: funcion_que_pide_memoria_dos_veces (in
/home/lucas/Escritorio/2019 Mendez algo2/ejemplos_valgrind/pruebas)
==3571==    by 0x40057C: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3571==
==3571== LEAK SUMMARY:
==3571==    definitely lost: 12 bytes in 3 blocks
==3571==    indirectly lost: 0 bytes in 0 blocks
==3571==      possibly lost: 0 bytes in 0 blocks
==3571==    still reachable: 0 bytes in 0 blocks
==3571==         suppressed: 0 bytes in 0 blocks
==3571==
==3571== For counts of detected and suppressed errors, rerun with: -v
==3571== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

En este ejemplo, si vemos el heap sumary, podemos notar en que contextos se pidió memoria, también nos pueden llegar a dar una pista de donde se termina perdiendo memoria, por ejemplo funcion_que_pide_memoria_dos_veces() resulta ser una función void, por lo que no devuelve nada, si no devuelve nada, no recibe parámetros, y encima pierde memoria significa que esa función debe ser arreglada. Por otro lado ¿por qué no decimos que pedir_memoria() está mal? la razón es simple, si bien pide memoria y no la libera, esta función devuelve un puntero a ese bloque de memoria por lo que todavía se posee referencia y no se perdió necesariamente.

## 4.4. Ejemplo 4

En el siguiente ejemplo vamos a mostrar un ejemplo donde se pierde memoria dos veces, con una sutileza, vamos a trabajar con un puntero doble.

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

int** ejemplo_puntero_doble(){
        int** numero=malloc(sizeof(int*));
        int*  segundo_puntero_numero=malloc(sizeof(int));
        *numero=segundo_puntero_numero;
        return numero;
}

int main(){
  int** numero=ejemplo_puntero_doble();
  **numero=8;
  return 0;
}
```

Si ejecutamos valgrind nos va a generar el siguiente mensaje:

```
    ==3749== Memcheck, a memory error detector
==3749== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3749== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3749== Command: ./pruebas
==3749==
==3749==
==3749== HEAP SUMMARY:
==3749==     in use at exit: 12 bytes in 2 blocks
==3749==   total heap usage: 2 allocs, 0 frees, 12 bytes allocated
==3749==
==3749== 4 bytes in 1 blocks are indirectly lost in loss record 1 of 2
==3749==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3749==    by 0x400545: ejemplo_puntero_doble (in /home/lucas/Escritorio/2019
Mendez algo2/ejemplos_valgrind/pruebas)
==3749==    by 0x40056C: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3749==
==3749== 12 (8 direct, 4 indirect) bytes in 1 blocks are definitely lost in loss
record 2 of 2
==3749==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3749==    by 0x400537: ejemplo_puntero_doble (in /home/lucas/Escritorio/2019
Mendez algo2/ejemplos_valgrind/pruebas)
==3749==    by 0x40056C: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3749==
==3749== LEAK SUMMARY:
==3749==    definitely lost: 8 bytes in 1 blocks
==3749==    indirectly lost: 4 bytes in 1 blocks
==3749==      possibly lost: 0 bytes in 0 blocks
==3749==    still reachable: 0 bytes in 0 blocks
==3749==         suppressed: 0 bytes in 0 blocks
==3749==
==3749== For counts of detected and suppressed errors, rerun with: -v
==3749== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

podemos notar que los 8 bytes de memoria perdidos son los correspondientes al espacio reservado para el puntero al numero (o sea el puntero doble), mientras que el espacio indirectamente perdido corresponde a el espacio reservado para el entero (o sea el puntero al entero).

## 4.5. Ejemplo 6

EL siguiente ejemplo es similar al anterior, solo que en este caso se realizo un free.

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

int** ejemplo_puntero_doble(){
        int** numero=malloc(sizeof(int*));
        int*  segundo_puntero_numero=malloc(sizeof(int));
        *numero=segundo_puntero_numero;
        return numero;
}

int main(){
  int** numero=ejemplo_puntero_doble();
  **numero=8;
  free(*numero);
  return 0;
}
```

podemos notar que acá estamos liberando un espacio de memoria, pero resulta ser insuficiente. viendo el informe de valgrind podemos ver que es lo que pasa realmente.

```
    ==2497== Memcheck, a memory error detector
==2497== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2497== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2497== Command: ./pruebas
==2497==
==2497==
==2497== HEAP SUMMARY:
==2497==     in use at exit: 8 bytes in 1 blocks
==2497==   total heap usage: 2 allocs, 1 frees, 12 bytes allocated
==2497==
==2497== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2497==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2497==    by 0x400577: ejemplo_puntero_doble (in /home/lucas/Escritorio/2019
Mendez algo2/ejemplos_valgrind/pruebas)
==2497==    by 0x4005AC: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==2497==
==2497== LEAK SUMMARY:
==2497==    definitely lost: 8 bytes in 1 blocks
==2497==    indirectly lost: 0 bytes in 0 blocks
==2497==      possibly lost: 0 bytes in 0 blocks
==2497==    still reachable: 0 bytes in 0 blocks
==2497==         suppressed: 0 bytes in 0 blocks
==2497==
==2497== For counts of detected and suppressed errors, rerun with: -v
==2497== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Acá podemos notar que todavía nos falta borrar un bloque de memoria, de 8 bytes,correspondientes al puntero que alocamos en memoria pero nunca borramos.

### 4.6. Ejemplo 7

Este ejemplo es similar al anterior, solo que esta vez se libera el otro puntero.

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

int** ejemplo_puntero_doble(){
        int** numero=malloc(sizeof(int*));
        int*  segundo_puntero_numero=malloc(sizeof(int));
        *numero=segundo_puntero_numero;
        return numero;
}

int main(){
  int** numero=ejemplo_puntero_doble();
  **numero=8;
  free(numero);
  return 0;
}
```

EL informe de valgrind nos va a producir el siguiente mensaje.

```
    ==2910== Memcheck, a memory error detector
==2910== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2910== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2910== Command: ./pruebas
==2910==

==2910== HEAP SUMMARY:
==2910==     in use at exit: 4 bytes in 1 blocks
==2910==   total heap usage: 2 allocs, 1 frees, 12 bytes allocated
==2910==
==2910== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2910==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2910==    by 0x400585: ejemplo_puntero_doble (in /home/lucas/Escritorio/2019
Mendez algo2/ejemplos_valgrind/pruebas)
==2910==    by 0x4005AC: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==2910==
==2910== LEAK SUMMARY:
==2910==    definitely lost: 4 bytes in 1 blocks
==2910==    indirectly lost: 0 bytes in 0 blocks
==2910==      possibly lost: 0 bytes in 0 blocks
==2910==    still reachable: 0 bytes in 0 blocks
==2910==         suppressed: 0 bytes in 0 blocks
==2910==
==2910== For counts of detected and suppressed errors, rerun with: -v
==2910== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

En este caso podremos ver que el bloque de memoria que se pierde es de 4 bytes, correspondientes a un int en una arquitectura de 64 bits.

## 5. Ejemplos memoria mal liberada

En la siguiente sección se van a mostrar algunos ejemplos donde hubo problemas al liberar memoria, ya sea por distintas razones.

## 5.1. Ejemplo 7

En este ejemplo resulta que un mismo bloque de memoria fue liberado más de una vez.

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

int* pedir_memoria(){
        int* numero=malloc(sizeof(int));
        return numero;
}

int main(){
  int* numero=pedir_memoria();
  *numero=8;
  free(numero);
  free(numero);
  return 0;
}
```

```
==3857== Invalid free() / delete / delete[] / realloc()
==3857==    at 0x4C2EDEB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3857==    by 0x4005B9: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3857==  Address 0x5204040 is 0 bytes inside a block of size 4 free'd
==3857==    at 0x4C2EDEB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3857==    by 0x4005AD: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3857==  Block was alloc'd at
==3857==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3857==    by 0x400577: pedir_memoria (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3857==    by 0x400593: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3857==
==3857== HEAP SUMMARY:
==3857==     in use at exit: 0 bytes in 0 blocks
==3857==   total heap usage: 1 allocs, 2 frees, 4 bytes allocated
==3857==
==3857== All heap blocks were freed -- no leaks are possible
==3857== For counts of detected and suppressed errors, rerun with: -v
==3857== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Si bien parece obvio en este ejemplo que no es correcto borrar más de una vez un mismo bloque de memoria, existen casos donde la aritmética de punteros pueda producir un error con los mismos mensajes. Otra cosa que se debe remarcar en este ejemplo es que en HEAP SUMMARY no nos advirtió de ningún bloque de memoria sin liberar, y eso es porque en realidad no se perdió ningún bloque de memoria, por lo que hay que asegurarse de no leer solo esa sección del informe de Valgrind.

## 5.2. Ejemplo 8

En el siguiente ejemplo va a ser una variación del Ejemplo 4.

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

int** ejemplo_puntero_doble(){
        int** numero=malloc(sizeof(int*));
        int*  segundo_puntero_numero=malloc(sizeof(int));
        *numero=segundo_puntero_numero;
        return numero;
}

int main(){
  int** numero=ejemplo_puntero_doble();
  **numero=8;
  free(numero);
  free(*numero);
  return 0;
}

```

Lo que podemos notar acá es que se realizan dos free, pero hay un detalle, el orden no es correcto. Si ejecutamos el programa con Valgrind, nos va a generar el siguiente informe.

```
==3024== Invalid read of size 8
==3024==    at 0x4005CE: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3024==  Address 0x5204040 is 0 bytes inside a block of size 8 free'd
==3024==    at 0x4C2EDEB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3024==    by 0x4005C9: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3024==  Block was alloc'd at
==3024==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3024==    by 0x400577: ejemplo_puntero_doble (in /home/lucas/Escritorio/2019
Mendez algo2/ejemplos_valgrind/pruebas)
==3024==    by 0x4005AC: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3024==
==3024==
==3024== HEAP SUMMARY:
==3024==     in use at exit: 0 bytes in 0 blocks
==3024==   total heap usage: 2 allocs, 2 frees, 12 bytes allocated
==3024==
==3024== All heap blocks were freed -- no leaks are possible
==3024==
==3024== For counts of detected and suppressed errors, rerun with: -v
==3024== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Acá podemos notar que no estamos perdiendo memoria, pero a su vez, estamos realizando una lectura invalida. para solucionar estos problemas, simplemente debemos invertir el orden de borrado.

## 5.3. Ejemplo 9

el siguiente ejemplo vamos a tratar de leer una posición de memoria que fue previamente liberado.

```c
#include<stdlib.h>
#include<stdio.h>
int* pedir_memoria(){
        int* numero=malloc(sizeof(int));
```

```
5         return numero;
6 }
7
8 int main(){
9   int* numero=pedir_memoria();
10  *numero=8;
11  free(numero);
12  printf("el numero es:%i\n",*numero );
13  return 0;
14 }
```

Ahora, veamos de nuevo la salida producida por valgrind.

```
==3240== Invalid read of size 4
==3240==    at 0x400602: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3240==  Address 0x5204040 is 0 bytes inside a block of size 4 free'd
==3240==    at 0x4C2EDEB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3240==    by 0x4005FD: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3240==  Block was alloc'd at
==3240==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3240==    by 0x4005C7: pedir_memoria (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3240==    by 0x4005E3: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3240==
el numero es:8
==3240==
==3240== HEAP SUMMARY:
==3240==     in use at exit: 0 bytes in 0 blocks
==3240==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==3240==
==3240== All heap blocks were freed -- no leaks are possible
==3240==
==3240== For counts of detected and suppressed errors, rerun with: -v
==3240== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Podemos notar que, aunque hayamos liberado el bloque de memoria, todavía pudimos obtener el valor dentro de el bloque de memoria, aun así, es incorrecto acceder a memoria ya liberada, porque ese espacio de memoria una vez liberado puede ser utilizado por otro proceso que es ejecutado al mismo tiempo.

## 5.4. Ejemplo 10

En este Ejemplo, vamos a escribir sobre una posición de memoria previamente liberada.

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

int* pedir_memoria(){
        int* numero=malloc(sizeof(int));
        return numero;
}

int main(){
  int* numero=pedir_memoria();
  *numero=8;
  free(numero);
  *numero=20;
  printf("el numero es:%i\n",*numero );
  return 0;
}
```

Ahora veamos la salida producida por valgrind.

```
==3296== Invalid write of size 4
==3296==    at 0x400602: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==  Address 0x5204040 is 0 bytes inside a block of size 4 free'd
==3296==    at 0x4C2EDEB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3296==    by 0x4005FD: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==  Block was alloc'd at
==3296==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3296==    by 0x4005C7: pedir_memoria (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==    by 0x4005E3: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==
==3296== Invalid read of size 4
==3296==    at 0x40060C: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==  Address 0x5204040 is 0 bytes inside a block of size 4 free'd
==3296==    at 0x4C2EDEB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3296==    by 0x4005FD: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==  Block was alloc'd at
==3296==    at 0x4C2DB8F: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3296==    by 0x4005C7: pedir_memoria (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==    by 0x4005E3: main (in /home/lucas/Escritorio/2019 Mendez
algo2/ejemplos_valgrind/pruebas)
==3296==
el numero es:20
==3296==
==3296== HEAP SUMMARY:
==3296==     in use at exit: 0 bytes in 0 blocks
==3296==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==3296==
==3296== All heap blocks were freed -- no leaks are possible
==3296==
==3296== For counts of detected and suppressed errors, rerun with: -v
```

debemos notar que el invalid read fue producido por la función printf, mientras que el invalid whrite por otro lado fue producido por la asignación a la variable *numero, finalmente podemos notar que se logró cambiar el número almacenado en nuestra variable.