Informe escrito Laboratorio #2 parte 1
Sistemas operativos

Estudiantes:

Brandon Duque García

Cristian Daniel Muñoz Botero

Profesor:

Danny Alexandro Múnera Ramírez

Departamento de Ingeniería de Sistemas

Facultad de Ingeniería

Universidad de Antioquia

2025

1. Se crea el programa pedido donde se crea un puntero a nulo y se trata de desreferenciarlo.

```
❯ nvim null.c
```

```
#include <stdio.h>

int main(){
    int* pointer = NULL;
    *pointer = 1;
}
```

Compila de manera correcta y no lanza ninguna advertencia ni error.

```
[cristian@my-arch lab02-memoria]$ gcc null.c -o null
[cristian@my-arch lab02-memoria]$
```

Al ejecutar el programa, se obtiene **Segmentation fault (core dumped)** ya que a la dirección que le tratamos de ingresar un valor es NULL.

```
[cristian@my-arch lab02-memoria]$ ls
null  null.c
[cristian@my-arch lab02-memoria]$ ./null
Segmentation fault (core dumped)
[cristian@my-arch lab02-memoria]$
```

2. Ponemos a debuggear el código

```
[cristian@my-arch lab02-memoria]$ gcc -g null.c -o null
[cristian@my-arch lab02-memoria]$ gdb null
GNU gdb (GDB) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from null...
(gdb) run
Starting program: /home/cristian/Documents/udea/semestre-8/sistemas-operativos/labs-so/lab02-memoria/null

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /lib64/ld-linux-x86-64.so.2
Downloading 9.66 M separate debug info for /usr/lib/libc.so.6
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555129 in main () at null.c:5
5           *pointer = 1;
(gdb)
```

El proceso de debuggeo muestra también una Segmentation fault. Específicamente la línea 5 donde estamos desreferenciando el puntero a entero con valor NULL.

3. Al llamar el programa valgrind con la flag de –leak-check=yes

```
[cristian@my-arch lab02-memoria]$ valgrind --leak-check=yes ./null
==6799== Memcheck, a memory error detector
==6799== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==6799== Using Valgrind-3.25.0 and LibVEX; rerun with -h for copyright info
==6799== Command: ./null
==6799==
==6799== Invalid write of size 4
==6799==    at 0x4001129: main (null.c:5)
==6799==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==6799==
==6799==
==6799== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==6799==  Access not within mapped region at address 0x0
==6799==    at 0x4001129: main (null.c:5)
==6799==  If you believe this happened as a result of a stack
==6799==  overflow in your program's main thread (unlikely but
==6799==  possible), you can try to increase the size of the
==6799==  main thread stack using the --main-stacksize= flag.
==6799==  The main thread stack size used in this run was 8388608.
==6799==
==6799== HEAP SUMMARY:
==6799==     in use at exit: 0 bytes in 0 blocks
==6799==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6799==
==6799== All heap blocks were freed -- no leaks are possible
==6799==
==6799== For lists of detected and suppressed errors, rerun with: -s
==6799== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
[cristian@my-arch lab02-memoria]$
```

Con la herramienta Valgrind se puede ver en la salida “Invalid write of size 4 … Address 0x0 is not stack’d, malloc’d or (recently) free’d”
También nos indica lo mismo que nos proporcionaba el gdb. (SIGSEGV): dumping core. Access not within mapped region at address 0x0.
Esto debido a que al parecer el puntero a NULL es lo mismo que ese puntero esté apuntando a la dirección 0x0, a la cual no se puede acceder porque no pertenece al proceso.

4. Escribimos un programa que reserve espacio en el heap con malloc pero no la libere antes de terminar el programa.

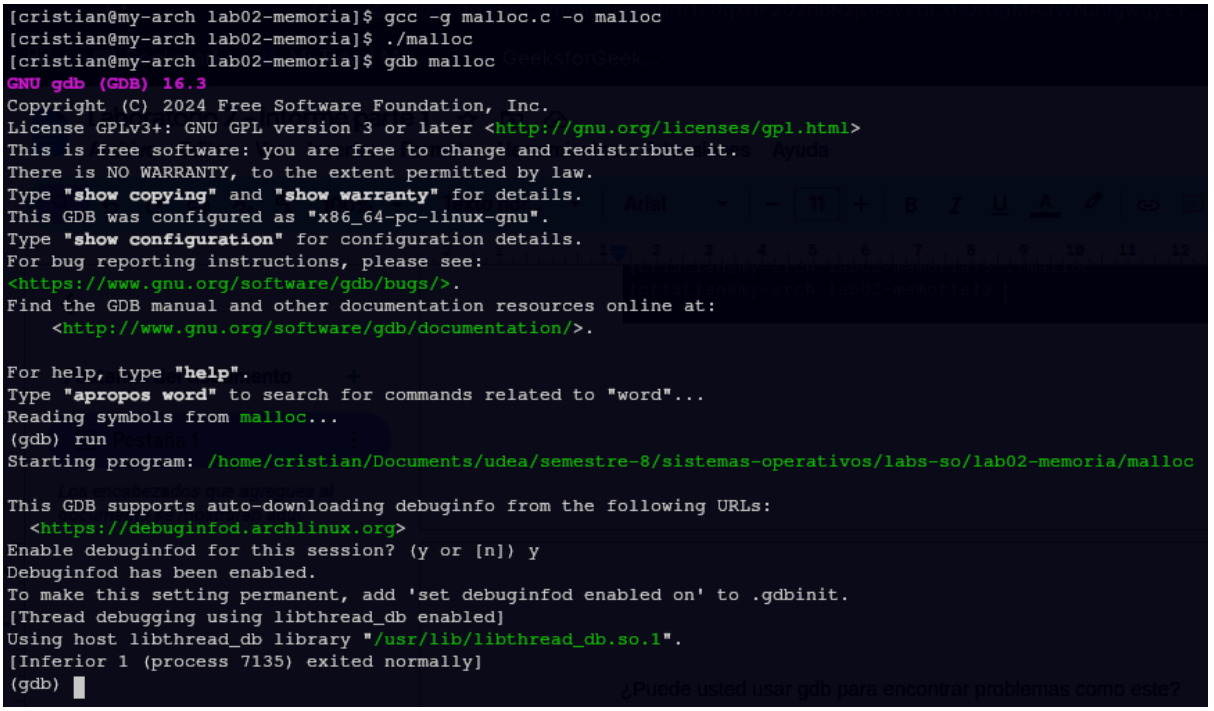
```
❯ nvim malloc.c
```

```
5 #include <stdlib.h>
4
3 int main(){
2     int* pointer = malloc(4*sizeof(int));
1 }
```

¿Qué pasa cuando el programa se ejecuta?: No pasa nada, el programa parece ejecutarse de manera correcta

```
❯ ~/Documents/udea/semestre-8/sistemas-operativos/labs-so/lab02-memoria
[cristian@my-arch lab02-memoria]$ gcc malloc.c -o malloc
[cristian@my-arch lab02-memoria]$ ./malloc
[cristian@my-arch lab02-memoria]$
```

¿Puede usted usar gdb para encontrar problemas como este?: No, ya que para gdb el programa también finaliza exitosamente sin dar problemas con el manejo de memoria, igualmente que cuando se ejecuta.



¿Que dice acerca de Valgrind?: Valgrind nos está indicando el memory leak que se presenta al no liberar la memoria. Primero nos da un reporte del uso del heap: total heap usage y luego nos indica 16 bytes in 1 blocks are definitely lost in loss record 1 of 1. Luego indica en el leak summary: definitely lost: 16 bytes in 1 blocks.
Esto indica que se perdió completamente el puntero. La memoria queda ocupada permanentemente durante la ejecución del programa, aunque ya no sea útil.

```
[cristian@my-arch lab02-memoria]$ valgrind --leak-check=yes ./malloc
==7162== Memcheck, a memory error detector
==7162== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==7162== Using Valgrind-3.25.0 and LibVEX; rerun with -h for copyright info
==7162== Command: ./malloc
==7162==
==7162==
==7162== HEAP SUMMARY:
==7162==     in use at exit: 16 bytes in 1 blocks
==7162==   total heap usage: 1 allocs, 0 frees, 16 bytes allocated
==7162==
==7162== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==7162==    at 0x484A7A8: malloc (vg_replace_malloc.c:446)
==7162==    by 0x400114A: main (malloc.c:4)
==7162==
==7162== LEAK SUMMARY:
==7162==    definitely lost: 16 bytes in 1 blocks
==7162==    indirectly lost: 0 bytes in 0 blocks
==7162==      possibly lost: 0 bytes in 0 blocks
==7162==    still reachable: 0 bytes in 0 blocks
==7162==         suppressed: 0 bytes in 0 blocks
==7162==
==7162== For lists of detected and suppressed errors, rerun with: -s
==7162== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[cristian@my-arch lab02-memoria]$
```

5. Creamos un programa que genera un array de 100 enteros con malloc

```
nvim array.c
```

```c
#include <stdlib.h>
int main(){
    int* data = malloc(100*sizeof(int));
    data[100] = 0;
    free(data);
    return 0;
}
```

Lo compilamos con la información simbólica para hacer después del debug, y el programa compila correctamente.

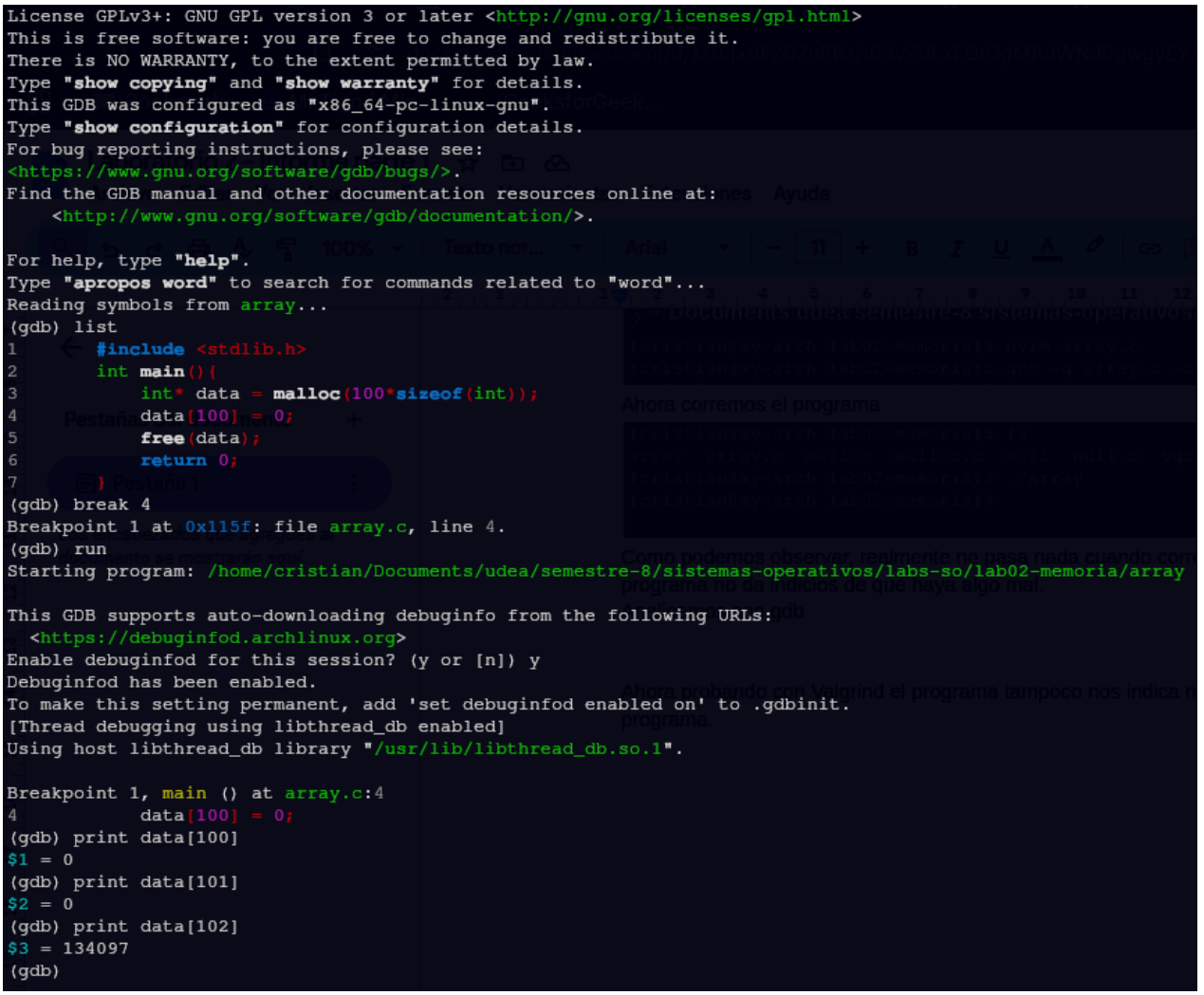
```
~/Documents/udea/semestre-8/sistemas-operativos/labs-so/lab02-memoria
[cristian@my-arch lab02-memoria]$ nvim array.c
[cristian@my-arch lab02-memoria]$ gcc -g array.c -o array
```

Ahora corremos el programa

```
[cristian@my-arch lab02-memoria]$ ls
array  array.c  malloc  malloc.c  null  null.c  vgcore.6799
[cristian@my-arch lab02-memoria]$ ./array
[cristian@my-arch lab02-memoria]$
```

Como podemos observar, realmente no pasa nada cuando corremos el programa. El programa no da indicios de que haya algo mal.

Analicemos con gdb



```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from array...
(gdb) list
1       #include <stdlib.h>
2       int main(){
3           int* data = malloc(100*sizeof(int));
4           data[100] = 0;
5           free(data);
6           return 0;
7       }
(gdb) break 4
Breakpoint 1 at 0x115f: file array.c, line 4.
(gdb) run
Starting program: /home/cristian/Documents/udea/semestre-8/sistemas-operativos/labs-so/lab02-memoria/array

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, main () at array.c:4
4           data[100] = 0;
(gdb) print data[100]
$1 = 0
(gdb) print data[101]
$2 = 0
(gdb) print data[102]
$3 = 134097
(gdb)
```

El programa parece correr normalmente, y por ejemplo nos permite acceder a las posiciones 101, y 102, y continuar. Esto por la creación del arreglo con malloc, el cual toma la base, la dirección del puntero data y empieza a sumar, por lo que en la memoria existirá la posición del arreglo 101, 102, y demás pero que se debe tener cuidado ya que no se definió que esas fueran posiciones del arreglo, por lo que tendrán datos basura, como se puede ver en data[102]. Por tanto el programa tiene un error grave de acceso a memoria ya que accede más allá del límite del arreglo y pese a que a veces parezca que funciona, es incorrecto y peligroso.

Ahora probando con Valgrind nos indica que hay un error en la línea 4 que es donde se asigna el valor 0 a data[100].

El mensaje de este error nos indica que la dirección 0x4a751d0 está después de un bloque de tamaño 400 que se aloja en 0x484a7a8 con un malloc que se realizó en la línea 3, donde creamos el array. Por lo que Valgrind está detectando que tenemos un error al asignar un valor a una posición mayor a su tamaño.

```
[cristian@my-arch lab02-memoria]$ nvim array.c
[cristian@my-arch lab02-memoria]$ gcc -g array.c -o array
[cristian@my-arch lab02-memoria]$ ./array
[cristian@my-arch lab02-memoria]$ valgrind --leak-check=yes ./array
==7961== Memcheck, a memory error detector
==7961== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==7961== Using Valgrind-3.25.0 and LibVEX; rerun with -h for copyright info
==7961== Command: ./array
==7961==
==7961== Invalid write of size 4
==7961==    at 0x4001169: main (array.c:4)
==7961==  Address 0x4a751d0 is 0 bytes after a block of size 400 alloc'd
==7961==    at 0x484A7A8: malloc (vg_replace_malloc.c:446)
==7961==    by 0x400115A: main (array.c:3)
==7961==
==7961==
==7961== HEAP SUMMARY:
==7961==     in use at exit: 0 bytes in 0 blocks
==7961==   total heap usage: 1 allocs, 1 frees, 400 bytes allocated
==7961==
==7961== All heap blocks were freed -- no leaks are possible
==7961==
==7961== For lists of detected and suppressed errors, rerun with: -s
==7961== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[cristian@my-arch lab02-memoria]$
```

Por lo tanto, el programa **NO** es correcto.

6. Creamos el programa requerido

```
❯ nvim array2.c
```

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

int main(){
    int* data = malloc(100*sizeof(int));
    //Incluimos datos en el arreglo
    for(int i = 0; i<100; i++){
        data[i] = i;
    }
    //Liberamos el arreglo
    free(data);

    printf("El valor data[50] es %d\n", data[50]);
}
```

El programa compila correctamente y también se ejecuta correctamente mostrando el valor que efectivamente tendría el arreglo en esa posición.

```
[cristian@my-arch lab02-memoria]$ gcc -g array2.c -o array2
[cristian@my-arch lab02-memoria]$ ./array2
El valor data[50] es 50
[cristian@my-arch lab02-memoria]$
```

Cuando usamos valgrind podemos obtener entonces un error en la línea 13 que es la línea donde estamos imprimiendo el valor del arreglo.
Nos indica que la dirección 0x4a75108 está dentro de un bloque de tamaño 400 liberado. Por lo que ya está disponible para nuevos datos y no se debería usar. El programa sería inseguro.

```
[cristian@my-arch lab02-memoria]$ valgrind --leak-check=yes ./array2
==9657== Memcheck, a memory error detector
==9657== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==9657== Using Valgrind-3.25.0 and LibVEX; rerun with -h for copyright info
==9657== Command: ./array2
==9657==
==9657== Invalid read of size 4
==9657==    at 0x40011B1: main (array2.c:13)
==9657==  Address 0x4a75108 is 200 bytes inside a block of size 400 free'd
==9657==    at 0x484D8EF: free (vg_replace_malloc.c:989)
==9657==    by 0x40011A6: main (array2.c:11)
==9657==  Block was alloc'd at
==9657==    at 0x484A7A8: malloc (vg_replace_malloc.c:446)
==9657==    by 0x400116A: main (array2.c:5)
==9657==
El valor data[50] es 50
==9657==
==9657== HEAP SUMMARY:
==9657==     in use at exit: 0 bytes in 0 blocks
==9657==   total heap usage: 2 allocs, 2 frees, 1,424 bytes allocated
==9657==
==9657== All heap blocks were freed -- no leaks are possible
==9657==
==9657== For lists of detected and suppressed errors, rerun with: -s
==9657== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[cristian@my-arch lab02-memoria]$
```

7. Creamos el programa requerido y ponemos el puntero funny aproximadamente en la mitad del arreglo que creamos anteriormente

```
nvim array2.c
```

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

int main(){
    int* data = malloc(100*sizeof(int));
    int* funny = &data[51];
    //Incluimos datos en el arreglo
    for(int i = 0; i<100; i++){
        data[i] = i;
    }
    //Liberamos el arreglo
    free(funny);

    printf("El valor data[50] es %d\n", data[50]);
}
```

Compilamos el programa e inmediatamente aparece un error en la compilación. Dice que a la función free se le pasó un puntero con el offset diferente de cero, por lo que no puede hacer la liberación ni compilar el programa. Esto simplemente con el gcc se puede detectar este problema, con el mismo compilador, no requerimos de gdb ni de Valgrind.

```
[cristian@my-arch lab02-memoria]$ nvim array2.c
[cristian@my-arch lab02-memoria]$ gcc -g array2.c -o array2
array2.c: In function 'main':
array2.c:12:5: warning: 'free' called on pointer 'data' with nonzero offset 204 [-Wfree-nonheap-object]
   12 |     free(funny);
      |     ^~~~~~~~~~
array2.c:5:17: note: returned from 'malloc'
    5 |     int* data = malloc(100*sizeof(int));
      |                 ^~~~~~~~~~~~~~~~~~~~~~~
[cristian@my-arch lab02-memoria]$
```

Con esto en mente, concluimos de este literal que es importante recordar siempre pasar a free() el mismo puntero que devolvió malloc.

8. Creamos el programa con una estructura básica y un método de push que usa realloc.

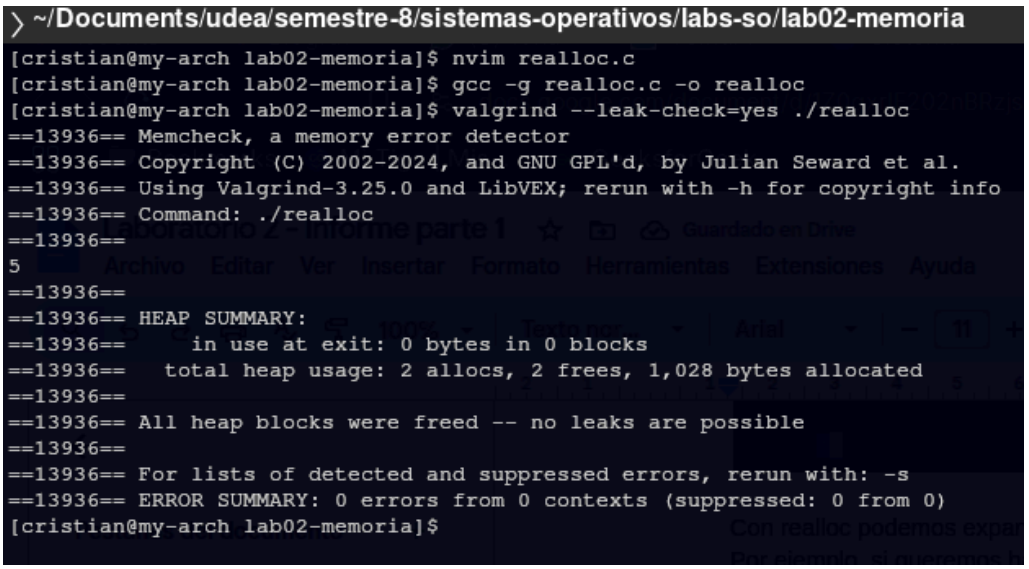
nvim realloc.c

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

typedef struct{
    int* data;
    size_t size;
} Vector;


void init(Vector* v) {
    v->data = NULL;
    v->size = 0;
}

void push(Vector *v, int value){
    v->data = realloc(v->data, (v->size+1) * sizeof(int));
    if(v->data == NULL){
        fprintf(stderr, "Error al asignar memoria\n");
        exit(1);
    }
    v->data[v->size] = value;
    v->size++;
}

void print(Vector *v) {
    for (size_t i = 0; i < v->size; ++i)
        printf("%d ", v->data[i]);
    printf("\n");
}

void free_vector(Vector *v) {
    free(v->data);
    v->data = NULL;
    v->size = 0;
}
```

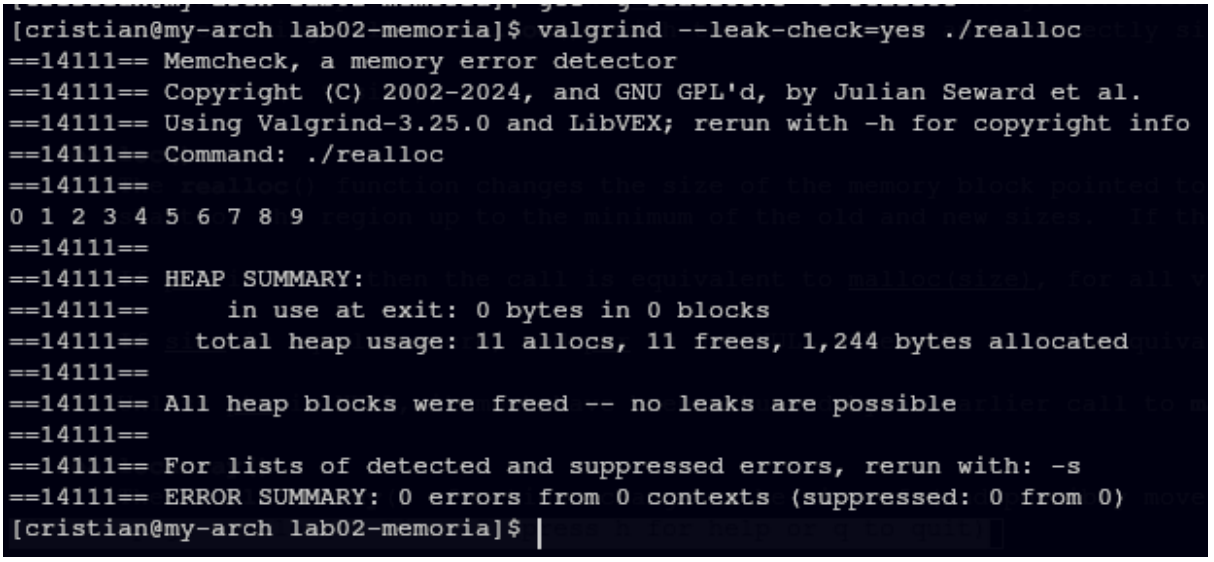
```c
int main(){
    Vector v;
    init(&v);

    push(&v, 5);
    print(&v);

    free_vector(&v);
    return 0;

}
```

Con realloc podemos expandir el espacio que teníamos anteriormente en data. Por ejemplo, si queremos hacer un push a un vector que tenía 2 enteros, entonces se hará el realloc([1, 2], 3*sizeof(int)), esto significa que el espacio de memoria que

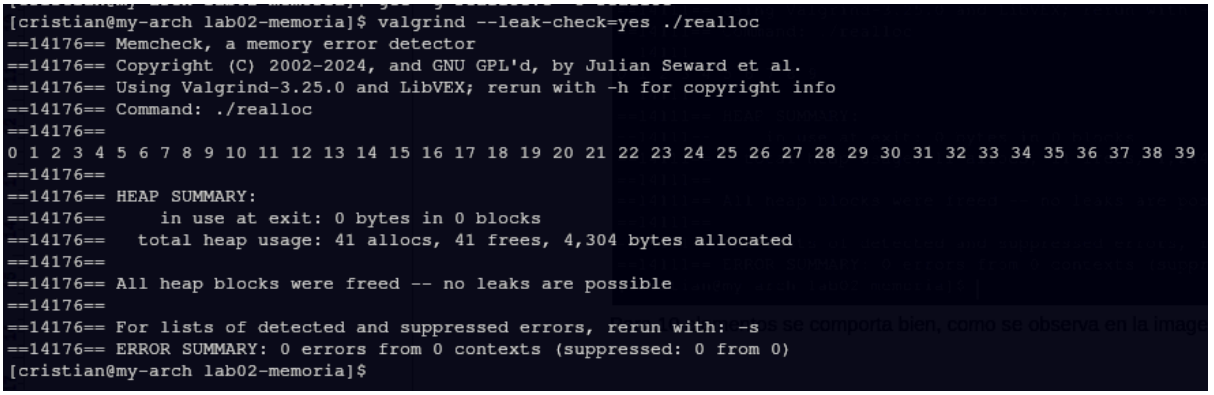
tenía data antes, que era del tamaño de 2 enteros, ahora será de 3 enteros y en este espacio, es donde se guarda el valor.
El arreglo, con un solo valor, se comporta correctamente cuando usamos valgrind:



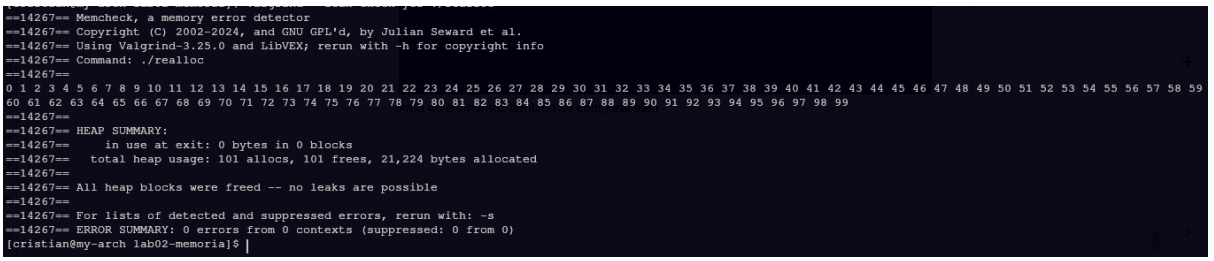
Agreguemos más elementos al vector:



Para 10 elementos se comporta bien, como se observa en la imagen anterior.



Para 40 elementos también se comporta bien



Lo mismo para 100 elementos.

El programa se comporta bien. El problema de esto está en el desarrollo, en la manera en la que se debe plantear la función de realloc para que se ejecute correctamente, la cual es más compleja que malloc. Realloc igualmente sigue siendo útil ya que mueve todo el bloque de memoria si en el espacio en el que está, no le es suficiente pero hay que tener en cuenta que este lo que hace es copiar esos datos cada vez que se llama, es decir cada vez que se pushea un elemento al arreglo, y esto a grandes iteraciones podría implicar en un alto uso de los recursos.

Una lista enlazada lo que hace es que cada nodo, apunta hacia su siguiente que no necesariamente está justo después de él en memoria física, puede estar atrás o adelante, lo que le da la forma son los enlaces entre los nodos. Pero un realloc lo que hace es expandir el espacio que tiene asignado un puntero, esto significa que los datos están todos juntos en memoria en la secuencia que deberían de seguir. Es un solo espacio grande de memoria.