Eines de depuració en C

Sistemes Operatius II – Grau en Enginyeria Informàtica – UB

Setembre 2016

Índex

Resum

Aquest document explica com es pot depurar un codi en C fent servir dues eines molt habituals per a aquests propòsits: el `gdb` (depurador) i el `valgrind`. Cal tenir en compte que només s'expliquen les principals funcionalitats que us poden ser útils. Les eines disposen d'un munt de funcionalitats addicionals a més de les descrites aquí. Si us interessen aneu al manual!

1 El depurador a C: `gdb`

En programar una aplicació ens trobarem sovint que aquest conté errors. Per tal de trobar aquests errors hi ha diverses formes de procedir. La primera és llegir el codi directament i imaginar l'execució del codi per tal d'intentar trobar en quin punt falla el codi. Una altra forma de procedir és imprimir per pantalla valors de variables o missatges per tal d'entendre quin és el flux d'execució de l'aplicació. Aquesta darrera forma és molt habitual per intentar trobar errors al nostre codi, tot i que realment no permet analitzar de forma precisa quin és el comportament de l'aplicació.

El depurador `gdb` és una aplicació de Linux que ens facilita analitzar i eliminar errors de la nostra aplicació ja que en permet estudiar el flux d'execució. L'aplicació `gdb` és una aplicació que funciona per línia de comandes. Existeixen però una sèrie d'aplicacions amb interfície gràfica que es comuniquen internament amb el `gdb` i que ens permeten utilitzar aquest depurador sense necessitat de fer servir la línia de comandes. Una d'aquestes aplicacions més conegudes és el DDD (Data Display Debugger).

En aquest document ens centrarem en la línia de comandes del `gdb`. La raó de procedir així és perquè es pugui entendre les facilitats que ens ofereix el depurador per analitzar el codi. A la majoria (per no dir tots) els depuradors amb interfície gràfica hi ha l'opció d'introduir la comanda a enviar al depurador. Per tant, tots els exemples mostrats a continuació es poden fer servir a qualsevol depurador que utilitzi el `gdb`.

1.1 Primers passos per depurar

Per tal de poder depurar el codi del nostre programa perquè el pugui fer servir el `gdb` cal compilar el codi perquè s'hi inclogui la informació necessària que el depurador necessita.

```
$ gcc -g programa.c -o programa
```

En cas que el nostre programa estigui format per múltiples fitxers cal compilar tots els fitxers amb l'opció "`-g`". No incloure l'opció "`-g`" implica que no podrem veure el codi font del nostre programa en depurar el nostre programa amb el depurador `gdb`.

Un cop haguem compilat el codi podem començar a depurar. Per tal de fer-ho hem d'executar el depurador des de línia de comandes passant com a argument el programa a depurar

```
$ gdb programa
```

Un cop s'hagi executat el depurador ens apareixerà per pantalla la línia de comandes d'aquest. Podem executar aleshores el nostre programa fent servir la comanda `run`

```
(gdb) run arg1 arg2
```

on `arg1` i `arg2` són dos arguments que volem passar al nostre programa. Per tal de poder analitzar el flux d'execució del nostre programa caldrà fer servir determinades comandes, que ens proporciona el depurador, per tal d'aturar el nostre programa en determinats punts que siguin del nostre interès i que ens permeten, per exemple, analitzar els valors de les variables.

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

int main(int argc, char **argv)
{
    int i, *a;

    a = malloc(-1); /* Malloc no valid */

    for(i = 0; i < 10; i++)
        a[i] = 0;

    return 0;
}
```

Figura 1: Codi `test1.c` del primer exemple de depuració, veure secció 1.2.

Observar que als exemples anteriors es fa servir el símbol "`$`" per indicar que la comanda s'executa des de terminal mentre que la cadena "`(gdb)`" es fa servir per indicar que la comanda s'introdueix per al depurador. Se seguirà aquesta notació al llarg d'aquest document.

Per tal de sortir del depurador es fa servir la comanda `quit`

```
(gdb) quit
```

Podem demanar ajuda al depurador respecte una comanda específica amb `help`

```
(gdb) help
```

1.2 Primer exemple de depuració

Comencem amb un primer exemple de depuració amb el fitxer `test1.c`, veure figura 1. És possible que reconeguem de seguida on és problema (línia 8). Aquí, però, l'objectiu és fer servir el depurador per trobar l'error.

Comencem per compilar el nostre programa incloent-hi la informació de depuració

```
$ gcc -g test1.c -o test1
```

Executem el nostre programa amb el depurador preferit. Aquí ho fem, insistim, amb el depurador per línia de comandes

```
$ gdb test1
```

A continuació executem el nostre programa i el depurador ens mostra un missatge d'error per pantalla

```
(gdb) run
Starting program: test1
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400579 in main (argc=1, argv=0x7fffffffddb8) at test1.c:11
11        a[i] = 0;
```

El depurador ens indica per pantalla la línia en què s'ha produït el problema (la línia 11) així com la comanda en què s'ha produït ("`a[i] = 0`").

En cas que estiguem fent servir directament el depurador `gdb` (cosa que no falta ja que disposem d'aplicacions amb interfícies molt amigables) podem passar a un mode de visualització

amigable amb la combinació de tecles "Control-x" seguida de la lletra "a". Això fa que es visualitzi a pantalla el codi font indicant el punt en què s'ha produït el problema (no es necessari, però, passar al mode amigable per fer els experiments que es descriuen més endavant). Disposem de la comanda `focus` per tal de seleccionar a quina part de la pantalla volem establir el focus.

Hem vist que el problema s'ha produït a la línia 11 del codi. Podem visualitzar els valors de les variables "`a`" i "`i`" amb

```
(gdb) print i
$1 = 0
(gdb) print a
$2 = (int *) 0x0
```

El problema es produeix a la primera iteració del bucle: el punter a la variable "`a`" és zero! En intentar accedir a un vector que no ha estat ubicat es produeix la violació de segment (en anglès, *segmentation fault*).

Anem a veure amb més detall si realment s'ubica el vector. Per això posarem un punt d'interrupció (en anglès, un *breakpoint*) a la línia en què es fa el *malloc*.

```
(gdb) break test1.c:8
Breakpoint 1 at 0x40054c: file test1.c, line 8.
(gdb) run
```

A l'exemple anterior indiquem que volem posar un punt d'interrupció a la línia 8 del fitxer `test1.c` i a continuació executem el nostre programa. Aquest s'aturarà just abans d'executar el *malloc*. Podem visualitzar el valor de "`a`" abans d'executar el *malloc*

```
(gdb) print a
$3 = (int *) 0x7fffffffddb0
```

Es tracta d'una variable no inicialitzada i el seu valor és aleatori: no és correcte suposar, al llenguatge C, que els valors de les variables s'inicialitzen per defecte a zero.

A continuació executem la comanda *malloc*. Per això fem

```
(gdb) next
(gdb) print a
$4 = (int *) 0x0
```

Amb la comanda `next` demanem executar la funció sense entrar-hi a dins (que es fa amb `step`, ja es comentarà més endavant). Amb aquest exemple podem veure doncs que en fer el *malloc* es retorna un punter zero. Això vol dir que la quantitat de memòria que hem demanat no és correcta. En concret, recordar que hem passat `-1` com a argument a la funció *malloc*.

1.3 Segon exemple de depuració

A continuació depurarem el codi `test2.c` de la figura 2. Anem a veure algunes comandes més per poder depurar. Observar que en complir-se la condició de la línia 11 s'assigna al punter un valor nul, línia 13. Aquí es posa manualment el punter a nul per tal de simular un problema en el codi.

Un cop compilat el nostre programa amb informació de depuració, l'executem amb el depurador

```
$ gcc -g test2.c -o test2
$ gdb test2
```

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

void funcio(int *a, int k)
{
    int i, j;

    j = 0;
    for(i = 0; i < 10; i++)
    {
        if ((i == 5) && (k == 2))
        {
            a = 0x0;
            j = 1;
        }

        a[i] = i;
    }
}

int main(int argc, char **argv)
{
    int *a;

    a = malloc(sizeof(int) * 10);

    funcio(a, 0);
    funcio(a, 1);
    funcio(a, 2);

    return 0;
}
```

Figura 2: Codi `test2.c` del primer exemple de depuració, veure secció 1.3.

I l'executem per tal de veure si podem treure alguna informació del problema que hi ha

```
(gdb) run
Starting program: test2
Program received signal SIGSEGV, Segmentation fault.
0x000000000040057c in funcio (a=0x0, k=2) at test2.c:17
13              a[i] = i;
```

El problema es produeix a la línia 17 del nostre programa. Podem demanar els valors les variables "`a`" i "`i`" amb

```
(gdb) print i
$1 = 5
(gdb) print a
$2 = (int *) 0x0
```

També podem demanar informació sobre la pila de funcions amb la comanda `info stack`

```
(gdb) info stack
#0  0x000000000040057c in funcio (a=0x0, k=2) at test2.c:17
#1  0x00000000004005da in main (argc=1, argv=0x7fffffffddb8) at test2.c:29
```

Observar a la figura 2 que *funcio* es crida tres cops a la funció *main*. La comanda anterior ens indica que el problema es produeix en cridar *funcio* a la línia 29 de *main*. Tingueu en compte que una funció es pot cridar des de diversos punts del codi. Per tant, és interessant poder saber des de quin punt del codi s'ha fet la crida que ha produït el problema.

Ara anem a utilitzar el depurador per fer posar un punt d'interrupció a la línia 29 del fitxer `test2.c`.

```
(gdb) break test2.c:29
Breakpoint 1 at 0x4005c9: file test2.c, line 29.
```

Executem el nostre programa

```
(gdb) run
```

L'aplicació s'aturarà a la línia en què es fa la crida a *funcio*. Recordar que podem passar a mode amigable amb la combinació de tecles "Control-x a".

Anem a executar *funcio* pas per pas. Per això executem

```
(gdb) step
```

Proveu què passa si executem `next` en comptes de `step`! Les comandes `next` i `step` són equivalents: permeten executar pas a pas les línies d'un codi. En cas que hi hagi una crida a una funció la comanda `next` executa tota la funció i passa la següent instrucció. En canvi, la comanda `step` permet entrar a l'interior de la funció per tal d'executar el seu codi.

Suposem que hem executat `step`. A continuació podem posar un punt d'interrupció a la línia 17, que és on es produeix el problema

```
(gdb) break test2.c:17
Breakpoint 2 at 0x400565: file test2.c, line 17.
```

Anem a veure algunes comandes que poden ser d'utilitat. En particular, ara anem a modificar el punt d'interrupció anterior perquè només s'hi aturi en cas que "`i = 5`" i "`a = 0x0`". Això ho fem amb

```
(gdb) condition 2 ((i == 5) && (a == 0x0))
```

El valor "2" indicat a la comanda anterior fa referència al punt d'interrupció número 2, que s'ha definit fa un moment. Ara continuem amb l'execució del nostre programa des del punt en què ens trobem actualment

```
(gdb) continue
Continuing.
Breakpoint 2, funcio (a=0x0, k=2) at test2.c:17
```

Ens hem aturat just en el punt en què es produeix el problema. Si ara fem un `next`...

```
(gdb) next
Program received signal SIGSEGV, Segmentation fault.
0x000000000040057c in funcio (a=0x0, k=2) at test2.c:17
```

Bingo! Ja sabem exactament on es produeix el problema. La pregunta és: en quin moment canvia de valor la variable "`a`"? En el codi que tenim queda evident, però suposem un codi molt més llarg en què no sapiguem en quin moment la variable "`a`" canviï de valor. Anem a veure com es pot fer. Per això esborrarem el punt d'interrupció 2 i tornem a executar

```
(gdb) delete 2
(gdb) run
(gdb) step
```

Ara tornem a ser a dins de la funció en què es produeix el problema. Amb la següent instrucció demanem que l'aplicació s'aturi en el moment en que la variable "`a`" canviï de valor

```
(gdb) watch a
Hardware watchpoint 3: a
```

Continuem l'execució de la nostra aplicació

```
(gdb) continue
Continuing.
Hardware watchpoint 3: a
Old value = (int *) 0x602010
New value = (int *) 0x0
funcio (a=0x0, k=2) at test2.c:14
```

El nostre programa s'ha aturat una instrucció després que el valor de la variable "`a`" hagi canviat de valor. Això ens permet analitzar la raó per la qual es produeix el problema i arreglar el problema. En particular, aquí ens diu que la variable "`a`" canvia de valor a la línia 13.

1.4 Altres instruccions d'interès

Com hem pogut veure, el depurador ens ofereix un conjunt d'instruccions que ens permeten analitzar de forma senzilla el flux d'execució d'un programa. Hi ha més instruccions que poden ser d'utilitat i aquí només se n'indiquen algunes:

- Modificar el valor de variables: es poden modificar els valors de les variables durant l'execució del nostre programa amb la comanda `set`. Per exemple, `set i = 3`, permet posar el valor de la variable "`i`" a 3.

- Executar fins finalitzar la funció actual: la comanda `finish` permet continuar l'execució del nostre programa fins que es retorni de la funció que s'està executant actualment. La funció mostrarà per pantalla el valor retornat per la funció.
- Llistar els punts d'interrupció: es pot veure un llistat dels punts d'interrupció que s'han definit amb `info breakpoints`.
- Punt d'interrupció per lectura de variables: la comanda `watch` aturarà el nostre programa en el moment en què la variable es modifiqui de valor; la comanda `rwatch` aturarà el nostre programa així que s'hi accedeixi per lectura; mitjançant la comanda `awatch` aturarà el nostre programa així que s'hi accedeixi per lectura o escriptura.

2 Depuració de memòria: `valgrind`

El `valgrind` és una aplicació per a Linux que permet detectar de forma ràpida i senzilla errors d'accessos a memòria. De forma similar al `gdb`, anem a veure aquí només algunes de les opcions que ens ofereix aquesta aplicació. Les opcions que es mostren aquí són suficients per realitzar les pràctiques de l'assignatura.

Atenció: és important mencionar que valgrind no tindrà un funcionament correcte en màquines virtuals. No el feu servir en màquines virtuals, tot i que sembli que funcioni correctament. A vegades també s'ha detectat un comportament no gaire correcte a sistemes Mac.

2.1 Motivació

Per tal de motivar la utilitat d'aquesta aplicació, comencem amb l'exemple mostrat a la figura 3. Analitzeu bé el codi i observeu que es realitzen accessos invàlids a memòria. Ens preguntem ara: compilarà el codi? En cas que compili, què passarà en executar-lo? Donarà algun error? Provem-ho!

```
$ gcc exemple1.c -o exemple1
$ ./exemple1
Abans assignacio
Despres assignacio
Valor a[15]: 15
$
```

Observar que el codi ha compilat i s'ha executat sense donar cap mena d'error. En llenguatge C no es comprova, en temps d'execució, si els accessos a vectors són dintre del rang correcte. Python i Java sí que ho fan: això pot alentir l'aplicació. En escriure fora de la memòria reservada potser estem sobreescrivint el valor d'altres variables que estan emmagatzemades en aquelles posicions de memòria. No ens fins més tard en l'execució que ens adonem que alguna cosa falla.

Podem provar d'executar aquest programa amb el depurador `gdb` i veurem que tampoc detecta cap problema. Com podem doncs detectar aquests tipus de problemes? Ho podem fer amb l'aplicació `valgrind`!

Perquè l'aplicació `valgrind` pugui ser capaç d'analitzar els problemes d'accés invàlids a memòria val compilar el nostre programa perquè s'hi inclogui la informació necessària que el depurador necessita.

```
$ gcc -g exemple1.c -o exemple1
```

Un cop compilat cal executar l'aplicació així

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

int main(void)
{
  int *a;

  a = (int *) malloc(sizeof(int) * 10);

  printf("Abans assignacio\n");
  a[15] = 15;
  printf("Despres assignacio\n");

  printf("Valor a[15]: %d\n", a[15]);

  free(a);
}
```

Figura 3: Codi `exemple1.c`, veure secció 2.1.

```
$ valgrind ./exemple1
```

L'aplicació `valgrind` comprova, en temps d'execució, si els accessos a memòria es realitzen correctament. La sortida de l'execució es mostra a la figura 4. Es pot veure que `valgrind` ens indica que hi ha una escriptura no vàlida a la línia 11 del codi, i que hi ha una lectura no vàlida a la línia 14 del codi.

Observar que aquesta aplicació no ens permet establir punts d'interrupció o fer una execució pas a pas com ho permet fer el depurador `gdb`. Però complementa molt bé el depurador ja que ens informa d'accessos invàlids. És important, però, comentar que `valgrind` ens permet detectar accessos invàlids a vectors ubicats amb memòria dinàmica, però no a la pila! Vegem un exemple...

2.2 Exemple amb memòria estàtica

Aquesta secció està dedicada a mostrar un exemple de problema que `valgrind` no es capaç de detectar. A la figura 5 es mostra un exemple, el codi exemple2.c, que fa servir memòria estàtica.

Provem d'executar l'aplicació des de línia de comandes

```
$ gcc exemple2.c -o exemple2
$ ./exemple2
Abans assignacio
Despres assignacio
Valor a[15]: 15
Violació de segment
$
```

Obervar que es produeix una violació de segment després de fer l'assignació i d'imprimir el valor de "`a[15]`" per pantalla. Ens podríem preguntar: on és produeix la violació de segment? Aquest codi ha estat escollit específicament per produir un determinat problema. En particular, recordar que a la pila s'emmagatzema informació, entre altres, sobre el punt de retorn d'una funció. En aquest cas, en escriure a la posició "`a[15]`" sobreescrivim el punt de retorn per a la funció *main* (que és el terminal). Hem sobreescrit aquesta informació i per això es produeix la violació de segment.

```
$ valgrind ./exemple1
==3657== Memcheck, a memory error detector
==3657== Copyright (C) 2002 2011, and GNU GPL'd, by Julian Seward et al.
==3657== Using Valgrind 3.7.0 and LibVEX; rerun with  h for copyright info
==3657== Command: ./exemple
==3657==
Abans assignacio
==3657== Invalid write of size 4
==3657==    at 0x400644: main (exemple1.c:11)
==3657==  Address 0x51d607c is not stack'd, malloc'd or (recently) free'd
==3657==
Despres assignacio
==3657== Invalid read of size 4
==3657==    at 0x40065C: main (exemple1.c:14)
==3657==  Address 0x51d607c is not stack'd, malloc'd or (recently) free'd
==3657==
Valor a[15]: 15
==3657==
==3657== HEAP SUMMARY:
==3657==     in use at exit: 0 bytes in 0 blocks
==3657==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==3657==
==3657== All heap blocks were freed    no leaks are possible
==3657==
==3657== For counts of detected and suppressed errors, rerun with:  v
==3657== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
$
```

Figura 4: Sortida de l'execució del programa `exemple1` amb `valgrind`.

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

int main(void)
{
  int a[10];

  printf("Abans assignacio\n");
  a[15] = 15;
  printf("Despres assignacio\n");

  printf("Valor a[15]: %d\n", a[15]);
}
```

Figura 5: Codi `exemple2.c` que fa servir memòria estàtica.

```
$ valgrind ./exemple2
==4244== Memcheck, a memory error detector
==4244== Copyright (C) 2002 2011, and GNU GPL'd, by Julian Seward et al.
==4244== Using Valgrind 3.7.0 and LibVEX; rerun with  h for copyright info
==4244== Command: ./exemple2
==4244==
Abans assignacio
Despres assignacio
Valor a[15]: 15
==4244== Jump to the invalid address stated on the next line
==4244==    at 0xF04E52455: ???
==4244==  Address 0xf04e52455 is not stack'd, malloc'd or (recently) free'd
==4244==
==4244==
==4244== Process terminating with default action of signal 11 (SIGSEGV)
==4244==  Bad permissions for mapped region at address 0xF04E52455
==4244==    at 0xF04E52455: ???
==4244==
==4244== HEAP SUMMARY:
==4244==     in use at exit: 0 bytes in 0 blocks
==4244==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==4244==
==4244== All heap blocks were freed    no leaks are possible
==4244==
==4244== For counts of detected and suppressed errors, rerun with:  v
==4244== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
$
```

Figura 6: Sortida de l'execució del programa `exemple2` amb `valgrind`. Observar que es mostren per pantalla una sèrie d'errors que són difícils d'entendre. En aparèixer aquests tipus d'errors cal sospitar que hi ha un problema a la pila.

Amb el `valgrind` no es poden detectar aquests tipus d'accessos invàlids, veure figura 6. Per això és recomanable utilitzar la memòria dinàmica (*malloc* i *free*) perquè `valgrind` pugui detectar els problemes d'accessos invàlids a memòria.

2.3 Detecció de memòria no alliberada

L'ús habitual (a les pràctiques) de `valgrind` és a) detectar accessos no vàlids, sigui per lectura o escriptura, a memòria dinàmica, b) realitzar operacions amb valors no inicialitzats a memòria dinàmica, c) detectar la memòria no alliberada. Vegem alguns exemples.

A la figura 7 es mostra el codi `exemple3.c`. Observar que en aquest exemple no alliberem la memòria en sortir.

Executem el nostre programa amb el `valgrind`, veure figura 8. Observar que `valgrind` ens indica que hi ha 40 bytes sense alliberar. També ens indica que hem de tornar a executar l'aplicació `valgrind` amb les opcions indicades en cas que vulguem més informació sobre les línies en què es fa la reserva de memòria. Executem `valgrind` amb les opcions que ens recomana, veure figura 9. Podem veure que `valgrind` ens indica a quines línies s'ha realitzat la reserva de memòria.

És important fer alliberar la memòria dinàmica? En sí el sistema operatiu allibera de forma automàtica tota la memòria dinàmica que un procés hagi reservat al llarg de l'execució del

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

int main(void)
{
  int *a, i;

  a = (int *) malloc(sizeof(int) * 10);

  for(i = 0; i < 10; i++)
    a[i] = 0;

  // No alliberem la memoria
}
```

Figura 7: Codi `exemple3.c` que es fa servir a la secció 2.3.

```
$ valgrind ./exemple3
==4971== Memcheck, a memory error detector
==4971== Copyright (C) 2002 2011, and GNU GPL'd, by Julian Seward et al.
==4971== Using Valgrind 3.7.0 and LibVEX; rerun with  h for copyright info
==4971== Command: ./exemple3
==4971==
==4971==
==4971== HEAP SUMMARY:
==4971==     in use at exit: 40 bytes in 1 blocks
==4971==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==4971==
==4971== LEAK SUMMARY:
==4971==    definitely lost: 40 bytes in 1 blocks
==4971==    indirectly lost: 0 bytes in 0 blocks
==4971==      possibly lost: 0 bytes in 0 blocks
==4971==    still reachable: 0 bytes in 0 blocks
==4971==         suppressed: 0 bytes in 0 blocks
==4971== Rerun with  --leak-check=full to see details of leaked memory
==4971==
==4971== For counts of detected and suppressed errors, rerun with:  v
==4971== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
$
```

Figura 8: Sortida de l'execució del programa `exemple3` amb `valgrind`.

```
$ valgrind --leak-check=full ./exemple3
==5179== Memcheck, a memory error detector
==5179== Copyright (C) 2002 2011, and GNU GPL'd, by Julian Seward et al.
==5179== Using Valgrind 3.7.0 and LibVEX; rerun with  h for copyright info
==5179== Command: ./exemple3
==5179==
==5179==
==5179== HEAP SUMMARY:
==5179==     in use at exit: 40 bytes in 1 blocks
==5179==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==5179==
==5179== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5179==    at 0x4C2ABED: malloc (in /.../vgpreload_memcheck amd64 linux.so)
==5179==    by 0x40054D: main (exemple3.c:8)
==5179==
==5179== LEAK SUMMARY:
==5179==    definitely lost: 40 bytes in 1 blocks
==5179==    indirectly lost: 0 bytes in 0 blocks
==5179==      possibly lost: 0 bytes in 0 blocks
==5179==    still reachable: 0 bytes in 0 blocks
==5179==         suppressed: 0 bytes in 0 blocks
==5179==
==5179== For counts of detected and suppressed errors, rerun with:  v
==5179== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
$
```

Figura 9: Sortida de l'execució del programa `exemple3.c` amb `valgrind` i les opcions que ens ha recomanat executar per detectar els punts en que reservem la memòria.

programa. Així que en sí pot semblar que no és important. Cal tenir en compte, però, que el nostre programa pot reservar memòria dinàmica de forma iterativa sense alliberar-la en el moment en què ja no la necessita. D'aquesta forma l'ordinador es pot quedar "sense memòria" anant cada cop més lent. És important, doncs, que ens acostumem a alliberar la memòria dinàmica quan ja no fa falta. Això és necessari fer-ho al llenguatge C. No cal fer-ho a Python o Java ja que l'intèrpret ens allibera automàticament la memòria quan se n'adona que ja no la fem servir (una forma senzilla és comprovar que no hi ha cap variable que apunti a la zona de memòria dinàmica).

2.4 Connexió entre el `valgrind` i el `gdb`

Hem comentat abans que `valgrind` no ens permet establir punts d'interrupció o fer una execució pas a pas com ho permet fer el depurador `gdb`. Però complementa molt bé el depurador ja que ens informa d'accessos invàlids. Una de les opcions que ens ofereix `valgrind` és executar el depurador a través de valgrind. Anem a veure un exemple.

Agafem l'exemple de la figura 4. Recordem que la sortida és 4. L'aplicació ens ha indicat, durant l'execució, en quin moment es produeixen accessos invàlids. Observar que l'aplicació `valgrind` no s'ha aturat en el moment en què s'ha produït l'accés invàlid. Hi ha alguna forma d'aconseguir-ho? Sí, veiem-ho amb el següent exemple.

El codi exemple4.c es mostra a la figura 10. Observar que es realitzen accessos invàlids a memòria. Compilem i executem el codi

```
$ gcc exemple4.c -o exemple4
```

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

int main(void)
{
    int *a, i;

    a = (int *) malloc(sizeof(int) * 10);

    for(i = 0; i < 100; i++)
        a[i] = 0;

    free(a);
}
```

Figura 10: Codi `exemple4.c` de la secció 2.4.

```
$ ./exemple4
*** Error in `./exemple4': free(): invalid pointer: 0x000000000230f010 ***
*** Error in `./exemple4': malloc: top chunk is corrupt: 0x000000000230f040 ***
```

En aquest cas l'execució de l'aplicació dóna un problema en executar *free*. Segurament, en escriure fora del vector que hem reservat, estem sobreescrivint informació que la funció *malloc* fa servir per gestionar tots els vectors que han estat reservats. Per a més informació reviseu la darrera pràctica de sistemes operatius 1 en què es treballa aquest tema.

Observar però que en cap moment tenim un problema en realitzar les assignacions a la variable "a". Aquestes es poden detectar amb el `valgrind`

```
$ valgrind ./exemple4
```

El resultat de l'execució es pot veure a la figura 11. L'aplicació `valgrind` ens indica que el problema es produeix a la línia 11 del codi. Però, per a quins valors de "`i`" es produeix el problema? `Valgrind` ens permet solucionar aquest problema.

Per això hem de seguir els següents passos

1. Executar el `valgrind` amb les següents opcions

   ```
   valgrind --vgdb=yes --vgdb-error=0 ./exemple4
   ```

   veurem que el `valgrind` sembla que es quedi bloquejat al terminal.

2. Obrim un altre terminal, executem el depurador, i introduim la indicada

   ```
   $ gdb ./exemple4
   ...
   (gdb) target remote | vgdb
   ```

   Aquesta darrera instrucció connecta el depurador amb el `valgrind`.

3. En aquest moment podem introduir, des del depurador, punts d'interrupció, per exemple. En aquest exemple no ens interessa introduir punts de control sinó que només volem veure per a quins valors de "`i`" accedim a posicions no vàlides. Per això hem d'introduir

   ```
   (gdb) continue
   ```

```
$ valgrind ./exemple4
==9899== Memcheck, a memory error detector
==9899== Copyright (C) 2002 2013, and GNU GPL'd, by Julian Seward et al.
==9899== Using Valgrind 3.10.1 and LibVEX; rerun with  h for copyright info
==9899== Command: ./exemple4
==9899==
==9899== Invalid write of size 4
==9899==    at 0x4005C0: main (exemple4.c:11)
==9899==  Address 0x51d8068 is 0 bytes after a block of size 40 alloc'd
==9899==    at 0x4C290D0: malloc (in /.../vgpreload_memcheck amd64 linux.so)
==9899==    by 0x40059E: main (exemple4.c:8)
==9899==
==9899==
==9899== HEAP SUMMARY:
==9899==     in use at exit: 0 bytes in 0 blocks
==9899==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==9899==
==9899== All heap blocks were freed    no leaks are possible
==9899==
==9899== For counts of detected and suppressed errors, rerun with:  v
==9899== ERROR SUMMARY: 90 errors from 1 contexts (suppressed: 0 from 0)
```

Figura 11: Sortida de l'execució del programa `exemple4` amb `valgrind`.

Observar que s'introdueix "`continue`" i no pas "`run`" per executar del programa (el programa l'està executant `valgrind`).

4. Un cop introduïda aquesta instrucció el programa s'aturarà automàticament en accedir de forma invàlida a una posició no vàlida de memòria

```
(gdb) continue
Program received signal SIGTRAP, Trace/breakpoint trap.
0x00000000004005c0 in main () at exemple4.c:11
11              a[i] = 0;
(gdb) print i
$1 = 10
```

L'accés invàlid es produeix per a "`i = 10`".