Club Informatique

Débogage avec GDB et Valgrind

Par Arthur Desuert
& Sébastien Michelland

Sommaire

- Introduction
- Chasse aux bugs avec GDB
- Colmater les fuites avec Valgrind

Introduction

C’est quoi un bug ?

« *Défaut de conception ou de réalisation d'un programme informatique, qui se manifeste par des anomalies de fonctionnement de l'ordinateur.* »

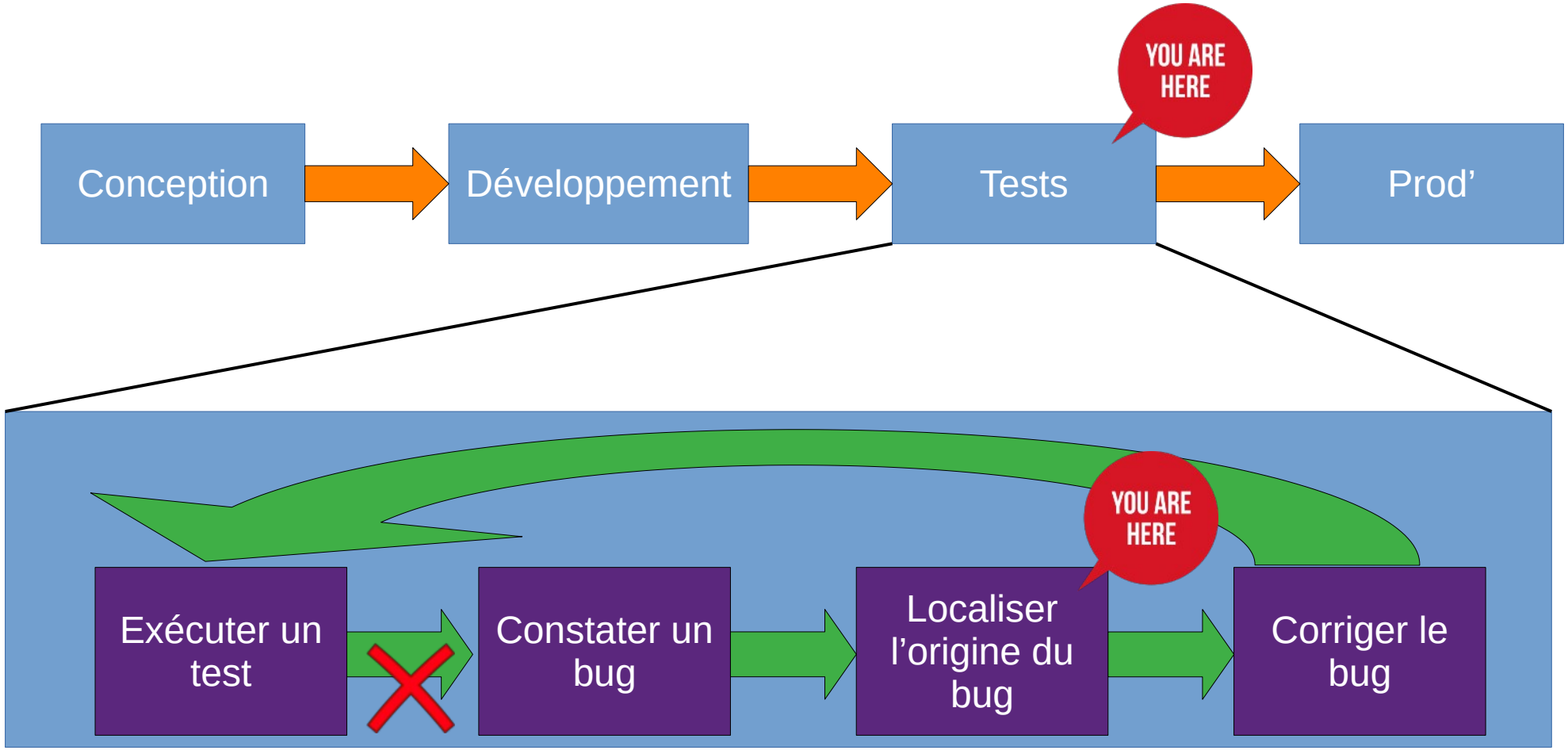
- Dictionnaire Larousse[1]

- Il est **impossible**[2] de prouver qu’un programme est sans bug

[1]https://www.larousse.fr/dictionnaires/francais/bogue/10005

[2]https://fr.wikipedia.org/wiki/Bug_(informatique)#Approche_formelle_:_les_m%C3%A9thodes_formelles

30 sec. de génie logiciel



Pourquoi utiliser un débogueur ?

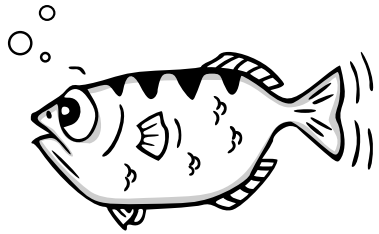
Méthode « printf » :

- ✗ Modifications parasites du code source
- ✗ (Re)compilations excessives
- ✗ Montre vite ses limites

Débogueur :

- ✔ Code source inchangé
- ✔ Fonctionnalités avancées
- ✔ Ca fait pro

GDB : GNU Debugger



- Développé par Richard Stallman en 1986.
- Gratuit et open-source (GPL)
- Supporte une 10^{aine} de langages (C, Go, Ada)
- Interface en ligne de commande

Valgrind

- Développé par Julian Seward
- Première release en 2002
- Gratuit et open-source (GPLv2)
- Pour les plateformes « UNIX-based » : Linux, Android, Darwin
- Référence à l’entrée principale de Valhalla

Fil conducteur

Un programme de manipulation de listes chaînées

- Ca compile !

```
user@gdb-session:~/gdb$ gcc main.c -o liste_chaine
user@gdb-session:~/gdb$
```

- What could possibly go wrong ?

Running into problems

- Quand on lance le programme…

```
user@gdb-session:~/gdb$ ./liste_chaine
Erreur de segmentation
```


Chasse aux bugs avec GDB

Compilation des sources

- Les compilateurs possèdent un flag « debug »
  - Pour GCC/Clang :

$ gcc/clang -g [...]

- Ajout d'infos de debug dans le binaire produit

```
user@gdb-session:~/gdb$ ls -l liste_chaine*
-rwxr-xr-x 1 user user 16776 oct.   8 14:04 liste_chaine
-rwxr-xr-x 1 user user 18240 oct.   8 14:27 liste_chaine_debug
```


Lancer GDB

- Depuis la ligne de commande :

$ gdb -q <exe>

```
user@gdb-session:~/gdb$ gdb -q liste_chaine_debug
Reading symbols from liste_chaine_debug...done.
(gdb)
```

- Depuis GDB (si déjà ouvert) :

(gdb) file <exe>

Obtenir de l’aide

- Dans GDB :

(gdb) help <commande>

- En ligne :

https://sourceware.org/gdb/current/onlinedocs/gdb/index.html

Lancer l'exécutable

- Lancement classique :

(gdb) run <args>

- Lancement et arrêt sur 1ère instruction :

(gdb) start <args>

Infos *post-mortem*

- GDB s'arrête à l'adresse qui a causé le SEGFAULT

```
Reading symbols from liste_chaine_debug...
(gdb) run
Starting program: /tmp/session_debug/liste_chaine_debug

Program received signal SIGSEGV, Segmentation fault.
0x000055555555551b5 in append_list (l=0x0, str=0x555555556004 "Hello gdb!") at main.c:22
22          while(l->next != NULL)
```

- Autres informations utiles :
  - La ligne de code fautive
  - La fonction courante et ses arguments

Infos *post-mortem*

- GDB n’est pas toujours si précis

```
(gdb) run
`/tmp/session_debug/liste_chaine_debug' has changed; re-reading symbols.
Starting program: /tmp/session_debug/liste_chaine_debug

Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:74
74      ../sysdeps/x86_64/multiarch/strlen-avx2.S: No such file or directory.
```

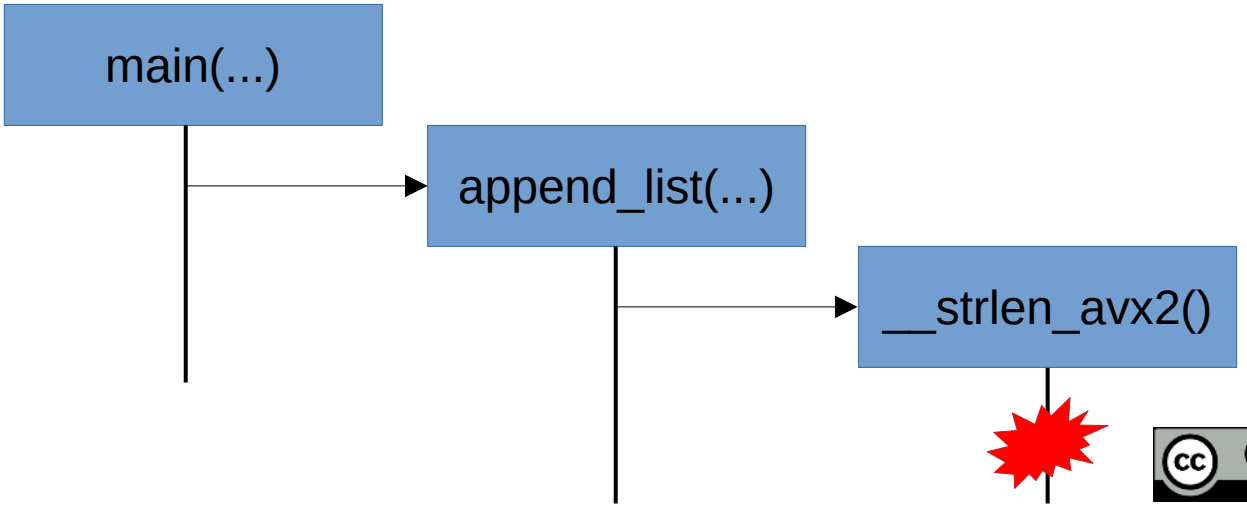
- Ici, appel à une fonction d’une librairie externe
  → pas d’accès au code source !

Pile d’appels

- GDB donne à la pile d’appels :

(gdb) backtrace

```
(gdb) backtrace
#0  __strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:74
#1  0x0000555555555183 in append_list (l=0x5555555592a0, str=0x0) at main.c:15
#2  0x000055555555524b in main () at main.c:50
```



Pile d'appels

- Pour naviguer dans cette pile :

(gdb) frame <num_frame>

```
(gdb) frame 1
#1  0×0000555555555183 in append_list (l=0×5555555592a0, str=0×0) at main.c:15
15          node→len = strlen(str);
```

- GDB rend actif la frame sélectionnée :
  - Affichage de la ligne en cours
  - Accès aux variables locales

Observer les variables (1)

- GDB peut afficher les variables locales d'une fonction :

(gdb) info locals

```
(gdb) info locals
node = 0×5555555592e0
```

- Pour affiche la pile d'appels + variables locales :

(gdb) backtrace full

Observer les variables (2)

- Afficher une variable précise :

(gdb) print <var>

Peut afficher chaînes de caractères, structures, etc.

```
(gdb) print node
$1 = (struct list *) 0x5555555592e0
(gdb) print *node
$2 = {len = 0, str = 0x0, next = 0x0}
```

- Affichage récurrent :

(gdb) display <var> / undisplay <var>

Faire une pause

- Placer un point d'arrêt **AVANT UNE INSTRUCTION** (*breakpoint*) :

  (gdb) break <cible>

- **<cible>** peut être :
  - Nom de fonction → **break *append_list***
  - Num. de ligne → **break *main.c:13***

Un break, et ça repart !

- Pour poursuivre l'exécution du programme :

(gdb) continue

- Pour avancer pas à pas :
  - En entrant dans les sous-routines :

(gdb) step

  - Sans entrer dans les sous-routines :

(gdb) next

Gestion des points d'arrêt

- Lister les points d'arrêt :

(gdb) info break

- Désactiver temporairement un point d'arrêt :

(gdb) disable <num_pt> / enable <num_pt>

- Supprimer un point d'arrêt :

(gdb) delete <num_pt>

Colmater les fuites avec Valgrind

Utilisation - memcheck

- Invocation classique (outil memcheck) :

$ valgrind <exe> <args>

- Analyse les erreurs mémoire au fil du programme
- Produit un rapport sur gestion de la mémoire allouée

Exemple de log memcheck

```
user@gdb-session:~/gdb$ valgrind ./liste_chaine_debug
==2049== Memcheck, a memory error detector
==2049== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2049== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==2049== Command: ./liste_chaine_debug
==2049==
==2049==
==2049== HEAP SUMMARY:
==2049==     in use at exit: 48 bytes in 2 blocks
==2049==   total heap usage: 3 allocs, 1 frees, 72 bytes allocated
==2049==
==2049== LEAK SUMMARY:
==2049==    definitely lost: 48 bytes in 2 blocks
==2049==    indirectly lost: 0 bytes in 0 blocks
==2049==      possibly lost: 0 bytes in 0 blocks
==2049==    still reachable: 0 bytes in 0 blocks
==2049==         suppressed: 0 bytes in 0 blocks
==2049== Rerun with --leak-check=full to see details of leaked memory
==2049==
==2049== For counts of detected and suppressed errors, rerun with: -v
==2049== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```


Chasse aux fuites

- Pour focaliser l'analyse sur la recherche de fuites mémoire :

  $ valgrind --leak-check=full <exe> <args>

- Remonte les lignes où sont alloués les blocs « perdus » à la fin de l'exécution

```
==2090== 24 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2090==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==2090==    by 0x10916E: append_list (main.c:12)
==2090==    by 0x109222: main (main.c:48)
```


Chasse aux erreurs mémoire

- Memcheck détecte également les erreurs de gestion de la mémoire
  - Accès hors bornes
  - Utilisation d'un pointeur après free()
  - Double free()

Merci de votre attention !

Bonne chasse aux bugs !

Références

- GNU Debugger, https://fr.wikipedia.org/wiki/GNU_Debugger, Wikipédia
- Valgrind, https://en.wikipedia.org/wiki/Valgrind, Wikipédia
- Software bug, https://en.wikipedia.org/wiki/Software_bug, Wikipédia
- **The Art of Debugging**, Norman Matloff and Peter Jay Salzman, ISBN-13 : 978-1-59327-174-9