

Programmation Structurée S6/S7

Cours 2 : GDB, Valgrind, etc

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/>

Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

septembre 2010



Le débogueur gdb

- 1 Le débogueur gdb
 - Introduction
 - Commandes de gdb

- 2 Le débogueur mémoire valgrind
 - Deux exemples

Présentation de gdb

- 1 Le débogueur gdb
 - Introduction
 - Commandes de gdb

- 2 Le débogueur mémoire valgrind

Source : Antoine Miné

GDB = débogueur interactif permettant de :

- interrompre et reprendre l'exécution du programme,
- suivre l'exécution du programme pas à pas,
- poser des points d'arrêt,
- inspecter le contenu des variables,
- connaître la ligne exacte et le contenu des variables au moment d'un `Segmentation fault`.

Source de l'exemple : Anne Canteaut.

Lancement de gdb

Préparation du programme

- compiler son programme avec l'option `-g`,
- éviter les options d'optimisation `-O2`, `-O3`,...

Exemple : `gcc toto.c -g -Wall -Wextra`

Lancement de gdb Dans le shell :

Tapez : `$> gdb a.out`

`gdb` propose un *shell* interactif, d'invite (`gdb`).

Running example

Nous allons déboguer le code fourni en annexe : `exemple.c`, dont la spécification est :

- lire deux matrices entières dont les tailles et les coefficients sont fournis dans un fichier `entree.txt`,
- calculer le produit
- afficher ce produit.

Exécution du programme sous gdb

Lancement du programme :

`$> (gdb) run`

L'exécution se termine ou est suspendue dès que :

- le programme **se termine** normalement :
`Program exited with code XXX`
- le programme **se termine** sur une erreur fatale :
`Program received signal SIGSEGV, Segmentation fault`
- l'utilisateur tape **contrôle+C** :
`Program received signal SIGINT, Interrupt`
- le programme passe par un **point d'arrêt** :
`Breakpoint X, fun at file : line`

Points d'arrêt

Placement d'un point d'arrêt :

`$> (gdb) break nom de fonction`

`$> (gdb) break num de ligne`

Suspend l'exécution à chaque fois que le programme :

- entre dans la fonction indiquée, ou
 - arrive au début de la ligne indiquée.
- on peut alors entrer de nouvelles commandes `gdb`.

Remarques

- on peut placer des points d'arrêt avant `run`,
- on peut placer plusieurs points d'arrêt,
- un point d'arrêt reste actif jusqu'à sa destruction explicite :
`delete` efface *tous* les points d'arrêt.

Reprise de l'exécution

Reprise de l'exécution `$> (gdb) continue`

Possible uniquement si l'exécution n'est que suspendue :

Possible

- après contrôle+C,
- sur un point d'arrêt.

Impossible

- avant `run`,
- après terminaison normale,
- après terminaison sur erreur.

Note : `run` recommence l'exécution au début.

Exécution pas à pas

Exécution d'une ligne

`$> (gdb) next`
`$> (gdb) step`

Effet : exécute une seule ligne et rend la main.

L'exécution peut être suspendue avant la ligne suivante :

- par contrôle+C,
- en cas de point d'arrêt,
- en cas d'appel de procédure pour `step`.

Contraintes : identiques à celles de `continue`.

Voir aussi : `finish`, `next n`, `step n`.

Inspection des données

Affichage d'une expression :

`$ > (gdb) print expr`

Expressions autorisées :

- opérateurs C classiques, y compris déréférences `*`, `[]`,
- variables globales,
- variables locales de la fonction en cours d'exécution.

Contraintes : le programme doit :

- soit être suspendu (point d'arrêt, `next`, etc.),
- soit s'être terminé sur une erreur fatale.

Inspection de la pile

Inspection de la pile : `$> (gdb) bt` (concis)
`$> (gdb) bt full` (détaillé)

Effet : affiche la pile d'appel et les arguments des fonctions.

Déplacement dans la pile

On peut se “déplacer” dans la pile d’appel pour choisir une fonction :

```
$> (gdb) up    (monte vers l'appelant)
$> (gdb) down  (descend vers l'appelé)
```

Effet :

- sur `print`
indique de quelles variables locales on parle.
- sur `finish` :
indique la fonction après laquelle `gdb` rend la main.

Aucun effet sur l’exécution du programme (`continue`, `next`,...).

Récapitulatif des commandes gdb

<code>quit</code> (ou <code>Ctrl+D</code>) <code>help</code>	quitte aide intégrée
<code>run</code> <code>continue</code> <code>next</code> (ou <code>next n</code>) <code>step</code> (ou <code>step n</code>) <code>finish</code>	commence l’exécution reprend l’exécution exécute une (ou <i>n</i>) ligne(s) " (mais s’arrête aux fonctions) exécute jusqu’au retour de la fonction
<code>Ctrl+C</code>	suspend l’exécution
<code>break num ligne</code> <code>break fonction</code> <code>delete</code>	place un point d’arrêt " efface tous les points d’arrêts
<code>print expr</code> <code>up</code> <code>down</code> <code>bt</code> (ou <code>bt full</code>)	affiche la valeur d’une expression remonte dans la pile d’appels descend dans la pile d’appels affiche la pile d’appels

Valgrind

- 1 Le débogueur `gdb`
- 2 Le débogueur mémoire `valgrind`
 - Deux exemples

Caractéristiques

- Débogueur mémoire, disponible pour Linux.
- Exécution « à la loupe » d’un binaire, en vérifiant les accès mémoire.
- Permet donc d’identifier des erreurs de programmation qui seraient passées inaperçues « par chance ».

Exemple 1

Premier programme (Source : Ensimag) :

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

char * s1 = "hello\n";

int main(void) {
    char * s2 = malloc(strlen(s1));
    strcpy(s2, s1);
    printf("%s", s2);
    return 0;
}
```

► On a oublié d'allouer un caractère pour '\0'.

Exemple 1 (suite)

Comme précédemment, on compile avec **-g**, puis :

```
valgrind ./expl-valgrind
```

On obtient parmi d'autres choses :

```
==13815== Invalid write of size 1
==13815==    at 0x401E9AD: strcpy (mc_replace_strmem.c:272)
==13815==    by 0x804840E: main (expl-valgrind.c:9)
==13815== Address 0x415302E is 0 bytes after a block of size 6 alloc'd
==13815==    at 0x401D38B: malloc (vg_replace_malloc.c:149)
==13815==    by 0x80483F7: main (expl-valgrind.c:8)
```

On vient de faire une écriture invalide d'un octet (« of size 1 ») à la ligne 9 de `expl-valgrind.c`, et l'adresse à laquelle on vient d'écrire se trouve just derrière un bloc alloué par `malloc` à la ligne 8 de `expl-valgrind.c`.

Exemple 2

Même source. La variable `s2` a été allouée (avec `malloc`), mais pas libérée (`free`) **fuite mémoire**

```
==13815== LEAK SUMMARY:
==13815==    definitely lost: 6 bytes in 1 blocks.
```

Pour en savoir plus ?

```
$ >valgrind --leak-check=full ./expl-valgrind
```

```
==4954== 6 bytes in 1 blocks are definitely lost in loss re
==4954==    at 0x4A05809: malloc (vg_replace_malloc.c:149)
==4954==    by 0x40055C: main (expl-valgrind.c:8)
```

► Donc ?

Documentation Valgrind

Pour aller plus loin (optimiser son code pour la gestion du cache, le faire tourner avec `gdb`, ...) :

<http://valgrind.org/>

► Documentation très détaillée avec des exemples !