

Debugger avec gdb

Laurent Réveillère

Enseirb
Département Télécommunications

`reveillere@enseirb.fr`
`http://www.enseirb.fr/~reveille/`

Constat

◆ Erreurs de programmation

□ Statiques

- » Détectées par le compilateur (validation syntaxique, typage)

□ Dynamiques

- » Comportement non souhaité (erreur sémantique)
- » Arrêt de l'exécution avec code d'erreur

◆ Corriger les erreurs

□ Statiques

- » Utiliser les informations de typage du compilateur

□ Dynamiques

- » Méthode du `printf`
 - Fastidieuse et ne marche que sur des exemples jouets
- » Inspection du programme pendant son exécution
 - Outil de débogage (Gdb, ddd, ...)

Messages d'erreurs classiques

- ◆ `Segmentation violation`
 - Tentative d'accès à une zone mémoire non autorisée
 - » Pointeur vers une zone non souhaitée
 - » Mauvais indice de tableau
- ◆ `Bus error`
 - Souvent causé par le déplacement dans une chaîne de caractères sans caractère de terminaison
- ◆ `Core dumped`
 - Sauvegarde par le système de l'état du programme en cours d'exécution dans un fichier appelé `core` dans le répertoire courant

Principes du débbuger

◆ Fonctionnement

- Inspecte le code assembleur
- Débogage symbolique si informations du code source ajoutées lors de la compilation

◆ Compilation sous gcc

- Utilisation de l'option `-g`
 - » Permet le débogage symbolique 😊
- Sinon
 - » Débogage de code assembleur 😞

◆ Exemple

```
gcc -g -o produit produit.c
```

Cas d'étude

- ◆ Récupérer les fichiers `produit.tar.gz` depuis <http://www.enseirb.fr/~reveille/Enseirb/IT105/produit.tar.gz>
- ◆ Extraire les fichiers `produit.c` et `entree.txt`
- ◆ Créer un nouveau projet
 - ❑ Isoler la fonction `main` dans le fichier `main.c`
 - ❑ Créer un fichier d'entête (`produit.h`) du module `produit`
 - ❑ Créer un fichier d'implémentation (`produit.c`) du module `produit`
 - ❑ Créer le fichier `Makefile` `ad-hoc`
- ◆ Compiler et exécuter le programme comme suit

```
./produit < entree.txt
```

Lancer gdb

- ◆ Depuis un shell

```
gdb <nom_executable>
```

- ◆ Dans emacs

```
M-x gdb <Enter> <executable>
```

Obligatoire !

- ◆ Une fois gdb lancé, l'invite de commandes est
(gdb)

- ◆ Recharger l'exécutable

- Erreur corrigée et exécutable modifié
- Pas nécessaire de relancer gdb

```
(gdb) file <executable>
```

Commandes utiles

◆ Quitter gdb

```
(gdb) quit
```

◆ Obtenir de l'aide

```
(gdb) help
```

```
[...]
```

```
(gdb) help files
```

```
[...]
```

```
(gdb) help quit
```

➔ Liste des classes de commandes

➔ Liste des commandes dans une classe

➔ Documentation d'une commande

◆ Raccourcis

- Beaucoup de commandes possèdent des formes abrégées
- Consulter l'aide

Premiers pas

◆ Lancer un programme

```
(gdb) run <arguments_du_programme>
```

◆ Arguments

- Comme depuis un shell

- Exemple

```
(gdb) run < entree.txt
```

- Aucun

 - » mêmes arguments que lors de la précédente session

◆ Voir les arguments mémorisés

```
(gdb) show args
```

◆ Exécution normale jusqu'au bout

```
Program exited normally.
```

```
(gdb)
```

Problème lors de l'exécution

- ◆ Informations sur la source de l'erreur
 - Nom de la fonction
 - Nom du fichier
 - Numéro de ligne
 - Type de l'erreur
- ◆ Cas de **gdb** lancé depuis Emacs
 - Emacs pointe sur la ligne où l'erreur a été détectée
- ◆ Refaire les dernière commandes à l'envers
 - Afficher les appels de fonction avec leurs arguments
 - Commande `backtrace`

Exécution pas à pas

◆ Objectif

- Exécuter une instruction ou une fonction à la fois
- Inspecter les valeurs du programme au fur et à mesure (comportement du printf)
- Permet d'identifier la source de l'erreur
- Possible de préciser un point du programme où s'arrêter

◆ Point d'arrêt

- Point du programme où gdb va stopper
- Le plus souvent numéro de ligne dans un fichier ou début d'une fonction
- D'autres formes plus compliquées existent

◆ Dans Emacs, se placer sur la ligne et taper `C-x <espace>`

◆ Dans gdb (`break` ou `b`)

```
(gdb) break nom_fichier::numéro de ligne
(gdb) b      nom_fichier::nom_fonction
```

Utilisation standard

1. Positionner un point d'arrêt sur la fonction main
2. Lancer l'exécution du programme
3. Inspecter l'état des variables (voir plus loin)
4. Exécuter les instruction pas à pas (voir plus loin)
5. Répéter les étapes 3 et 4 autant que nécessaire
6. Reprise de l'exécution normale avec la commande `continue`

Gestion des points d'arrêt

◆ Durée de vie

- Un point d'arrêt reste toujours actif

◆ Liste des points d'arrêts

```
(gdb) info breakpoints
```

◆ Suppression (delete ou d)

```
(gdb) delete <numero>
```

◆ Voir aussi les commandes

- `info` Obtenir de l'information sur un point d'arrêt
 (numéro, ...)
- `clear` Supprimer plusieurs points d'arrêts à la fois

Inspecter l'état du programme

- ◆ Afficher la valeur d'une expression

```
(gdb) print TAB[25][72]
```

```
(gdb) $1 = 5
```

- ◆ \$1 est une variable interne à **gdb** dont on peut se servir ensuite au lieu de taper `TAB[25][72]`
- ◆ Préciser le nom d'une variable si plusieurs occurrences
 - `nom_de_fonction::variable`
 - `'nom_de_fichier'::variable`
- ◆ Modifier la valeur d'une variable

```
(gdb) set variable <nom_variable> = <expression>
```

Valeur de retour d'un appel de fonction

- ◆ Invocation d'une fonction
 - Comme en C classique

```
(gdb) print <nom_fonction> (<arguments>)
```

- ◆ Valeur de retour
 - Affichée à l'écran

Exécution pas à pas

Processus permettant le plus souvent de trouver le bug !!!

- ◆ Exécuter une instruction à la fois (`step` ou `s`)

```
(gdb) step <argument>
```

- Descend dans le corps d'une fonction appelée
- Commande `finish` pour continuer l'exécution jusqu'à la fin de la fonction
- `step n` exécute `n` fois la commande `step`

- ◆ Passer à l'instruction suivante (`next` ou `n`)

```
(gdb) next <argument>
```

- `next n` exécute `n` fois la commande `next`
- Si l'instruction est une fonction, exécute toutes les instructions de la fonction appelée