INF280 : Préparation aux concours de programmation

Débogage de programmes C++

Antoine Amarilli

Avec un IDE (environnement de développement intégré)

- Eclipse CDT, Visual Studio Code, CLion…
- Fonctionnalités avancées (débogage, profilage mémoire, etc.)
- Lourd à installer, parfois lourd à utiliser
- Difficile à utiliser avec des éditeurs externes (emacs, vim…)
- Difficile pour des programmes non développés avec l'IDE

→ On présente des outils Linux qui ne dépendent pas d'un IDE

Traçage

- On peut toujours mettre des `printf` en C++
- Utiliser grep pour rapidement repérer l'information utile
- Utiliser les deux sorties (sortie standard, sortie d'erreur : `fprintf(stderr, ...)`)
- Ne pas oublier de supprimer le debug avant de soumettre !
- En Shell Unix :
  - `>` redirige la sortie standard vers un fichier
  - `2>` redirige la sortie d'erreur vers un fichier
  - `|` envoie cout vers cin d'un autre programme (p. ex., `less`, `grep`)
  - `|&` envoie cout et cerr dans l'entrée d'un autre programme

Remarque

Rappel : Pour la lecture de l'entrée du problème et l'écriture des résultats, il faut utiliser `scanf` / `printf` *et non* `cin` / `cout`.

Les bonnes options de compilation de GCC

`-Wall` pour des messages d'avertissement pour détecter des bugs évidents (variables non utilisées, retour de fonction manquant, = utilisé dans un if(), etc.) ; possible aussi `-Wextra`

`-g` pour inclure les symboles dans l'exécutable généré et permettre le débogage

`-std=c++11` pour utiliser C++ 2011

`-O2` à ne pas utiliser quand on veut déboguer : fait disparaître certaines variables ou instructions, etc.

Débogage mémoire

Programmes identifiant les problèmes de mémoire, débordement de pile, accès à de la mémoire non allouée sur le tas… :

valgrind est une machine virtuelle exécutant le programme en contrôlant chacun des accès mémoire ; très lent, mais très efficace

```
valgrind ./mon_programme
```

Très utile pour comprendre l'origine d'un segfault, ou si le programme semble faire n'importe quoi. À essayer tôt.

Permettent en général d'identifier les bugs au moment où ils apparaissent (mais uniquement pour la mémoire dynamique), contrairement à gdb en cas de corruption mémoire

GDB : utilisation

- On lance GDB avec `gdb ./mon_programme`
- Interfaces graphiques :

gdb -tui	pour une simili-interface graphique, assez pratique
ddd, KDbg, Insight	interfaces graphiques plus ou moins complètes

Principales commandes de GDB

r < input	pour démarrer le programme en lisant `input` sur sa sortie standard
r	redémarre le programme
b nom_fonction	pour positionner un point d'arrêt
clear	supprime le point d'arrêt en cours
c	continue jusqu'au prochain point d'arrêt
n, s	avance d'une instruction (mais s va à l'intérieur des appels de fonction)
bt, u, d	affiche et navigue dans la pile des appels
print expr	affiche la valeur d'une expression
watch expr	arrêtera le programme quand la valeur de l'expression changera
q	quitte GDB

Example de compilation et exécution

- Pour tester avec toutes les optimisations :

```
g++ -Wall -Wextra -std=c++11 -O2 problem.c &&
  ./a.out < input
```

- Le `&&` évite de laisser passer une erreur de compilation
- Pour déboguer :

```
g++ -Wall -Wextra -std=c++11 -g problem.c &&
  gdb ./a.out
r < input
```

- Pour interrompre l'exécution n'importe où : CTRL+C

Crédits

- Version initiale de ces transparents par Pierre Senellart