# INF280 : Projet de programmation : Problèmes pratiques et concours

Débogage de programmes C++

Pierre Senellart, Antoine Amarilli 21 février 2017

## Avec un IDE (environnement de développement intégré)

- Eclipse CDT, NetBeans, Visual C++, XCode, KDeveloper...
- · 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
- ightarrow On présente des outils Linux qui ne dépendent pas d'un IDE

#### Traçage

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

#### Remarque

Pour la lecture de l'entrée du problème et l'écriture des résultats, cin / cout est parfois trop lent, privilégier dans ces cas scanf / printf, voire read / write

## Les bonnes options de compilation de GCC

- -Wall -Wextra pour des messages d'avertissement pour détecter des bugs évidents (variables non utilisées, retour de fonction manquant, = utilisé dans un if(), etc.)
  - -g pour inclure les symboles dans l'exécutable généré et permettre le débogage
  - -std=c++11 pour utiliser C++ 2011
    - -02 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.

**efence** redéfinit les fonctions d'allocation mémoire pour détecter les accès incorrects au tas

LD\_PRELOAD=libefence.so ./mon\_programme

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 -pedantic -std=c++11 -O2 problem.c &&
    ./a.out < input</pre>
```

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

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

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