Mettre au point un programme

Lorsqu'un programme ne fonctionne pas comme prévu, on peut souvent détecter des erreurs en l'exécutant en pas-à-pas (une ligne de code après l'autre) et en examinant l'évolution du contenu des variables. L'application qui permet de faire cela s'appelle un déboggueur[1]. Plusieurs interfaces existent, on présente ici `gdb` sous `emacs` qui s'exécute dans l'environnement d'édition du texte source.

1. Utilisez l'option `-g` lorsque vous compilez (pour utiliser le menu Compile d'emacs, modifiez la première ligne du texte source comme par exemple:
   `// -*- mode:C++; compile-command:"g++ -g -Wall essai.cc" -*-`)

2. Compilez le programme, on suppose dans la suite que le programme compilé s'appelle `a.out` ce qui est le cas par défaut

3. Dans `emacs`, sélectionnez Debugger (GDB) dans le menu Tools (ou tapez sur la touche `Echap`, vous devez voir apparaitre `ESC-` au bout de quelques secondes, puis selon la version d'emacs tapez `xgud-gdb` ou `xgdb` et la touche `Entree`). Vous devez alors voir dans la ligne d'état en bas:
   `Run gdb (like this): gdb a.out`
   Modifiez si nécessaire puis tapez `Entree`. Le deboggueur est activé.

4. Pour visualiser à la fois le source du programme et le débbugueur, il peut être nécessaire de couper la fenêtre en deux (menu File->Split Window ou raccourci clavier Ctrl-X 2), et dans l'une des parties de sélectionner le source (menu Buffers ou raccourci Ctrl-X o).

5. L'étape suivante consiste à mettre un *point d'arrêt* près de l'endroit où vous suspectez qu'il y a une erreur (par exemple au début d'une fonction suspecte). Il y a deux méthodes, soit en donnant le nom de la routine, par exemple pour le programme principal `main`:
   `b main`
   soit dans le code source, en cliquant à gauche de la ligne ou en tapant `Ctrl-X` puis `Ctrl-A` puis `Ctrl-B` à la ligne souhaitée. On peut aussi taper `b` suivi par le numéro de ligne dans le fichier source (ou `b nom_ficher:numero_ligne`). Vous pouvez mettre plusieurs points d'arrêt.

6. Puis on lance l'exécution du programme en cliquant sur Go ou en tapant `r` (pour run)

7. Le programme s'exécute normalement jusqu'à ce qu'il atteigne le point d'arrêt. On peut alors visualiser le contenu d'une variable en utilisant l'instruction `p` suivi du nom de variable (p pour print), par exemple:
   `p modulo` puis touche `Entree`
   imprime le contenu de la variable `modulo` (si elle existe).

8. L'instruction `n` (next) exécute la ligne courante. Pour exécuter ensuite plusieurs lignes à la suite, on tape plusieurs fois sur la touche Entree. On peut aussi indiquer un paramètre numérique à la commande `n`.

[1] Pour des erreurs d'allocation mémoire, on utilisera plutôt un logiciel spécialisé comme `valgrind`

9. L'instruction `s` (step in) permet de rentrer à l'intérieur d'une fonction si la ligne courante doit exécuter cette fonction (alors que `n` exécute cette fonction en un seul pas, sauf si cette fonction contient un point d'arrêt)

10. L'instruction `u` (until) exécute le programme sans arrêt jusqu'à ce que la ligne de code source suivante soit atteinte (ou une adresse donné en paramètre). C'est particulièrement intéressant pour sauter au-delà d'une boucle.

11. L'instruction `c` (continue) permet de continuer l'exécution jusqu'au prochain point d'arrêt. On peut aussi indiquer un paramètre pour ignorer le point d'arrêt un certain nombre de fois.

12. `kill` permet de stopper le programme en cours,

13. `d` permet de détruire un point d'arrêt (on donne alors son numéro) ou tous les points d'arrêt. On peut aussi temporairement désactiver ou réactiver un point d'arrêt (commandes `disable` et `enable`)

14. `f 0`, `f 1`, etc. permet de visualiser la fonction appelante à l'ordre demandé, l'ordre est calculé depuis la fonction interrompue (0 pour la fonction actuelle, 1 pour la 1ère appelante, etc.), ainsi que les variables de cette fonction. Cette commande peut d'ailleurs être utilisée en cas de Segmentation fault sans mettre de point d'arrêt au préalable.

15. Vous pouvez corriger une erreur dans la fenêtre d'édition, recompiler le programme et revenir dans la fenêtre `gdb` (c'est le buffer nommé `*gud-a.out*`) puis reprenez ce mode d'emploi à l'étape 5 (relancer l'exécution du programme en tapant `r`, etc.).

16. Pour quitter le déboggueur, tapez `q` (quit) ou fermez le buffer `*gud-a.out*` (activez ce buffer et choisissez `Close` dans le menu `Files`)

Remarques:

- Les variables de type structuré ou les classes s'affichent comme des structures C, même si vous avez redéfini l'opérateur `<<`. Pour afficher une variable de ce type, il est judicieux de prévoir une méthode:
  `void dbgprint() const { cout << *this << endl; }`
  On peut alors visualiser une variable `x` du type considéré en tapant
  `print x.dbgprint()`
  Pour automatiser tout cela, créez un fichier nommé `.gdbinit` et contenant
  ```
  echo Defining v as print command for class types\n
  define v
  print ($arg0).dbgprint()
  end
  ```
  Dans la session `gdb`, tapez `v x` pour visualiser le contenu de la variable `x`.

- Parmi les autres possibilités intéressantes de gdb, il y a la modification du contenu d'une variable en cours d'exécution (commande `print variable=valeur`) On peut aussi interrompre le programme seulement si une condition est vérifiée (`break .. if ..`).

- Pour plus d'informations, vous pouvez taper `help` dans la fenêtre du debuggueur pour obtenir l'aide en ligne ou à tout moment dans `emacs`, en tapant `Ctrl-H I`, puis `Ctrl-S gdb` puis clic de la souris avec le bouton du milieu sur `gdb`.