

Outils d'étude et de débogage

Philippe Collet
2007-2008

D'après des cours de L. Pierre et R. Rousseau

- SPLint
- gdb, ddd
- valgrind

Lint, Lclint, SPLint

- Principe
 - Outil d'analyse statique pour détecter des *anomalies* dans du code source C
- Historique
 - Lint dans les années 70, puis LcLint et SPLint (GNU)
- Anomalies détectées
 - Variables non utilisées ou utilisées avant leur initialisation
 - Affectations dans les conditionnelles
 - Boucle infinie, code non atteignable
 - Fonctions ou arguments non utilisés
 - Types incompatibles (par exemple sur les valeurs retournées)

Principes du débogage

- Enquêter pour trouver l'origine d'un problème
- Équiper le code de traceurs et de la capacité à s'arrêter et reprendre :
 - Démarrer le programme en ayant défini ce qui pourrait modifier son comportement
 - Arrêter le programme à des endroits spécifiques, selon certaines conditions
 - Examiner ce qui s'est passé, les valeurs des variables, etc.
 - Modifier certaines choses dans le programme ou des valeurs, afin d'expérimenter et de corriger l'effet d'une erreur.

gdb

- Débogueur (GNU) en ligne
 - Le programme C doit avoir été compilé avec l'option `-g`
- Lancement et arrêt
 - `gdb nomDeLExecutable`
 - `run ARG`
 - exécuter le programme avec les arguments ARG
 - `quit`
 - pour quitter gdb

Des points d'arrêt

- Ajout
 - **break FUNCTION**
 - arrêt à l'entrée de la fonction FUNCTION
 - **break LINENUM**
 - arrêt sur la ligne numéro LINENUM
 - **break POSITION if CONDITION**
 - arrêt sur la POSITION (une fonction ou un numéro de ligne) si la condition CONDITION est vraie
- Gestion
 - **info break**
 - donne la liste des points d'arrêt
 - **disable/enable BREAKNUM**
 - désactiver/activer un point d'arrêt

Des points d'arrêt (suite)

- Suppression des points d'arrêt
 - **clear FUNCTION**
 - supprime le point d'arrêt sur la fonction FUNCTION
 - **clear LINENUM**
 - supprime le point d'arrêt sur la ligne LINENUM
 - **delete**
 - supprime tous les points d'arrêt
 - **delete BREAKNUM**
 - supprime le point d'arrêt numéro BREAKNUM de la liste obtenue avec info break

Variables et fonctions

- Surveillance de variables
 - **watch VAR**
 - arrêt lorsque la variable VAR change de valeur
 - **watch CONDITION**
 - arrêt lorsque la CONDITION est vraie
- Affichages d'information
 - **print VAR**
 - imprime le contenu de la variable VAR
 - **print &VAR**
 - imprime l'adresse de la variable VAR
 - **whatis VAR**
 - donne le type de la variable VAR
 - **whatis FUNCTION**
 - donne le type de la fonction FUNCTION
 - **where**
 - affiche le contenu de la pile (fonctions)

Navigation et modifications

- Navigation
 - **continue**
 - exécuter jusqu'au prochain point d'arrêt
 - **step**
 - instruction suivante (entrer dans le code d'une fonction)
 - **next**
 - instruction suivante (sans entrer dans le code d'une fonction)
 - **display VAR**
 - affiche à chaque pas le contenu de VAR
- Modification
 - **set VAR=VALUE**
 - affecte la valeur VALUE à la variable VAR

ddd

- Principe
 - ddd est une interface graphique pour gdb et d'autres débogueurs
 - on y retrouve les mêmes fonctionnalités.
 - Le programme C doit toujours avoir été compilé avec l'option -g.
- Fonctionnalités
 - le fichier contenant le main s'affiche dans le cadre du haut
 - la console gdb s'affiche dans le cadre du bas
 - la fenêtre de commandes DDD (command tool) est affichée

P. Collet 9

Fenêtre principale

The screenshot shows the initial DDD window with the following components labeled:

- Argument Field**: Located at the top left of the source window.
- Source Window**: The main area displaying the C source code.
- Debugger Console**: The bottom area showing the GDB output.
- Status Line**: The bottom-most line showing the current status.
- Command Tool**: A vertical toolbar on the right side of the source window.
- Scroll Bar**: A vertical scrollbar on the right side of the source window.

Initial DDD Window

Exécuter le programme

The screenshot shows the DDD interface with a breakpoint set on the line `a = (int *)malloc(C);`. A dialog box titled "Run with Arguments" is open, showing the arguments `0000 2000 0030 1800 0000`. Labels indicate:

- Breakpoint**: Points to the red dot on the source code line.
- Click here to run**: Points to the "Run" button in the dialog box.
- Arguments**: Points to the text field in the dialog box.

Running the Program

Observer des valeurs

The screenshot shows the DDD interface with the execution position at the line `a = (int *)malloc(C);`. A tooltip (Value Tip) is displayed over the variable `a`, showing its memory address and value: `(int *) 00004000: 00004000`. Labels indicate:

- Execution Position**: Points to the red dot on the source code line.
- Value Tip**: Points to the tooltip showing the value of `a`.

Viewing Values in DDD

Modifier une valeur

The screenshot shows a debugger window with a source code editor. A dialog box titled 'Set Value' is open, allowing the user to change the value of a selected variable. The variable 'size' is selected in the source code, and the dialog box shows its current value and a field to enter a new value. Labels 'Set Button', 'Select variable in the source', and 'Edit value' point to the 'Set Value' button, the source code, and the input field in the dialog box, respectively.

Setting a Value

valgrind

- valgrind permet de détecter certains types d'erreur
 - sur des machines d'architecture x86 sous Linux
 - En licence GPL
 - Essentiellement des **erreurs de gestion de mémoire**
- Principe
 - toutes les lectures/écritures en mémoire sont vérifiées
 - tous les appels aux fonctions malloc et free sont interceptés
- Les erreurs détectables sont donc
 - utilisation de mémoire non initialisée
 - lectures/écritures dans des espaces mémoire préalablement libérés par free
 - lectures/écritures en dehors des espaces mémoire alloués par malloc
 - passage à des appels système de blocs mémoire non initialisés ou non adressables
 - ...
- Pour l'utiliser, il suffit d'exécuter le programme en cause, en précisant qu'on l'exécute sous valgrind
- **Aucune action préalable n'est nécessaire.**

P. Collet 14

Conclusion

Comprendre le cycle de vie du logiciel

P. Collet 15

Les phases du cycle de vie

The diagram shows the phases of the software lifecycle in a circular flow: Objectifs, Définition des besoins, Analyse des besoins, Planification, Conception, Implémentation et tests unitaires, Validation et Intégration, Qualification, Mise en exploitation, Maintenance, and Retrait ou remplacement.

16

Objectifs

- Fixés par *les donneurs d'ordre*
 - le management
 - ou une (bonne) idée...
- Quelques définitions
 - Clients : ceux qui veulent le produit
 - Utilisateurs : ceux qui vont l'utiliser
 - Développeurs : ceux qui vont le fabriquer

Définition des besoins

- Un cahier des charges est normalement établi par le **client** en interaction avec utilisateurs et encadrement :
 - description des fonctionnalités attendues
 - contraintes non fonctionnelles (temps de réponse, place mémoire,...)
 - possibilités d'utilisation de *Use Cases*
- ☞ *A l'issue de cette phase : cahier des charges*

Analyse des besoins

- C'est la définition du produit
 - Spécification précise du produit
 - Contraintes de réalisation
- A l'issue de cette phase :
 - Client et fournisseur sont d'accord sur le produit à réaliser (IHM comprise)
 - ☞ *Dossier d'analyse (spécifications fonctionnelles et non fonctionnelles)*
 - ☞ *Ébauche de manuel utilisateur*
 - ☞ *Première version du glossaire du projet*

Planification

- Découpage du projet en tâches avec enchaînement
 - Affectation à chacune d'une durée et d'un effort
 - Définition des normes qualité à appliquer
 - Choix de la méthode de conception, de test...
 - Dépendances extérieures (matériels, experts...)
- ☞ *Plan qualité + Plan projet (pour les développeurs)*
- ☞ *Estimation des coûts réels*
- ☞ *Devis destiné au client (prix, délais, fournitures)*

Conception

- Définition de l'architecture du logiciel
- Interfaces entre les différents modules
- Rendre les composants du produits indépendants pour faciliter le développement

- ☞ *Dossier de conception*
- ☞ *Plan d'intégration*
- ☞ *Plans de test*
- ☞ *Mise à jour du planning*

Implémentation et tests unitaires

- Codage et test indépendant de chaque module
- Produits intermédiaires :

- ☞ *Modules codés et testés*
- ☞ *Documentation de chaque module*
- ☞ *Résultats des tests unitaires*
- ☞ *Planning mis à jour*

Validation et Intégration

- Chaque module est intégré avec les autres en suivant le plan d'intégration
- L'ensemble est testé conformément au plan de tests

- ☞ *Logiciel testé*
- ☞ *Tests de non-régression*
- ☞ *Manuel d'installation*
- ☞ *Version finale du manuel utilisateur*

Qualification

- Tests en vraie grandeur, dans des conditions normales d'utilisation
- Tests non-fonctionnels :
 - Tests de charge
 - Tests de tolérance aux pannes
- Parfois Bêta-test
- ☞ *Rapports d'anomalie*
- *Déterminant dans la relation client-fournisseur*

Mise en exploitation

- Livraison finale du produit (packaging)
- Installation chez le client
- Est-ce la fin des problèmes ?

☞ **AU CONTRAIRE**

☞ **Ce n'est rien en comparaison de la...**

Maintenance

- Rapport d'incident (ou anomalie)
- Demande de modification corrective
- Demande d'évolution (avenant au contrat)
- Code et documentation modifiés...
- Nouvelle série de tests :
 - unitaires
 - d'intégration
 - de non-régression

Sources

- Cours de C avancé et Environnement de Programmation (L. Pierre, 2003-2004)
- Cours et TD de Génie Logiciel en Master 1 Info (R. Rousseau / P. Collet, 2004-2005)