Environnement de compilation et de mise au point.

Algorithmique 3 - Travaux pratiques

Il est demandé aux étudiants de réaliser chaque exercice dans un répertoire séparé.
Les travaux seront réalisés à partir de l'archive `base_code_lab1.tar.gz` fournie sur moodle et contenant la hiérarchie suivante :

`base_code_lab1`
–> `Test` : répertoire contenant les fichiers de test.
–> `Exercice1` : répertoire contenant les fichiers pour le premier exercice.
–> `Exercice2` : répertoire contenant les fichiers pour le second exercice.

Cette séquence ne fait pas l'objet d'un dépôt obligatoire mais, à la demande de l'enseignant, l'étudiant devra pouvoir répondre aux questions posées dans le sujet et fournir une archive similaire à l'archive de départ contenant le résultat de son travail.

1 Compilation séparée, tests et pré-conditions

Concepts étudiés :

- Compilation séparée et configuration des outils de compilation.
- Compilation en mode mise au point ou production.
- Notion de pré-condition sur les opérations, responsabilités de vérifications.

Compétences acquises dans cette partie :

- Savoir utiliser les différents modes de compilation pour le développement d'un logiciel.
- Savoir définir les responsabilités de vérification de pré-conditions sur les opérations.

Étant donnés les fichiers sources `stack.h`, `staticstack.c` et `main.c`, correspondant à l'implantation statique de la pile vue en cours, et à son programme de test il est demandé de réaliser le travail suivant :

1.1 Makefile et compilation

À partir du document *Introduction aux makefiles* disponible sur la page moodle du cours, écrire un fichier `Makefile` permettant de générer, lors de la saisie de la commande `make` le programme exécutable `stack_ex1` en mode production, optimisé par `-O3` et définissant le symbole `NDEBUG`.
Ce `Makefile` devra aussi permettre une compilation en mode mise au point (option `-g`) lors de la compilation par la commande `make DEBUG=yes`.
Un modèle d'un tel `Makefile` est décrit en page 9 du document *Introduction aux makefiles*.
Vous ajouterez à votre `Makefile` une étiquette `doc`, dépendant de `stack.h` et de `main.c` et permettant de générer la documentation de votre code en utilisant la commande :

```
doxygen documentation/TP1
```

Lorsque vous entrez la commande `make doc`, la documentation HTML de votre code sera accessible dans le fichier `documentation/html/index.html`.

1.2 Exécution du programme et préconditions

Les fichiers de test fournis contiennent, sur la première ligne le nombre d'éléments à ajouter à la pile et sur la ligne suivante, les éléments à ajouter à la pile en conservant l'ordre donné dans le fichier.

1. Exécuter le programme `stack_ex1` résultant de la compilation en mode production sur les jeux de tests fournis en lançant les commandes :

   ```
   ./stack_ex1 ../Test/testfile1.txt
   ```

   et

   ```
   ./stack_ex1 ../Test/testfile2.txt
   ```

2. En comparant l'affichage produit avec le contenu des fichiers de test, conjecturez le problème à l'origine de la différence entre l'affichage et les données lues.
3. Recompiler votre programme en mode mise au point, exécuter à nouveau le programme sur les jeux de test fournis. Qu'est-ce qui vous permet de vérifier votre conjecture précédente ?
4. Corriger le programme main.c de façon à ce qu'il renvoie le code d'erreur "1" si le test ne peut pas être exécuté.

2 Débogueur et analyse dynamique (valgrind)

Concepts étudiés :

— Outils d'analyse dynamique de programmes.
— Détection et correction d'erreurs de gestion mémoire.
— Détection et correction de gaspillage de ressources.

Compétences acquises dans cette partie :

— Savoir utiliser un débogueur pour l'analyse et la correction de programme.
— Savoir utiliser un profiler pour l'analyse et la correction de programme.

Lors du développement d'un logiciel, il est important de pouvoir s'assurer, d'une part de son bon fonctionnement, mais aussi de sa bonne utilisation des ressources. Les outils d'analyse dynamique permettent d'instrumenter et d'examiner un programme pendant son exécution. L'objectif de cet exercice et de vous faire prendre contact avec les outils d'analyse dynamique les plus répandus que sont les débogueur et les profiler de code.
Nous utiliserons pour cet exercice le débogueur `gdb` et le profiler `valgrind`.

`Gdb` est un logiciel GNU permettant de déboguer les programmes C (et C++). Il permet de répondre aux questions suivantes :

— à quel endroit s'arrête le programme en cas de terminaison incorrecte, notamment en cas d'erreur de segmentation ?
— quelles sont les valeurs des variables du programme à un moment donné de l'exécution ?
— quelle est la valeur d'une expression donnée à un moment précis de l'exécution ?

Gdb permet donc de lancer le programme, d'arrêter l'exécution à un endroit précis, d'examiner et de modifier les variables au cours de l'exécution et aussi d'exécuter le programme pas-à-pas.

Outre la documentation de référence de gdb [1], une introduction à l'utilisation de GDB est accessible sous moodle.

`Valgrind` est un ensemble d'outils d'instrumentation pour l'analyse dynamique de programme. Il existe des outils Valgrind capables de détecter automatiquement de nombreux bogues de gestion de la mémoire et de thread, et de profiler vos programmes en détail. La distribution Valgrind comprend actuellement six outils utilisés en production :

— un détecteur d'erreur de mémoire,
— deux détecteurs d'erreur de thread,
— un profileur de prédiction de cache et de branchement,
— un générateur de graphes d'appel et un profileur mémoire.

Outre la documentation de référence [2] les étudiants sont invités à consulter le guide de démarrage rapide de valgrind à l'url http://valgrind.org/docs/manual/quick-start.html

À partir du fichier `dynamicstack.c`, correspondant à l'implantation dynamique de la pile vue en cours et implantant l'interface définie dans `stack.h`, de votre fichier `main.c` et de votre `Makefile` que vous recopierez dans le répertoire `Exercice2`, il est demandé de réaliser le travail suivant :

2.1 Makefile et compilation

1. Modifier votre `Makefile` et votre programme principal pour utiliser le module `dynamicstack` et générer l'exécutable `stack_ex2`.
2. Compiler en mode production et exécuter les tests en lançant les commandes :

```
./stack_ex2 ../Test/testfile1.txt
```

et

```
./stack_ex2 ../Test/testfile2.txt
```

Que se passe-t-il lors de l'exécution ? Le débogueur vous renseigne-t-il sur l'erreur constatée ?

2.2 Débogage du code

1. Compiler votre programme en mode mise au point et exécutez votre programme sous le débogueur. Quel renseignement complémentaire obtenez-vous sur votre erreur ?
2. En entrant la commande suivante, exécutez votre programme sous valgrind.

```
valgrind ./stack_ex2 ../Test/testfile1.txt
```

Quel renseignement complémentaire obtenez-vous ?

3. Corrigez le problème identifié et vérifier le bon fonctionnement du programme sur les tests fournis.

1. Documentation de référence : https://sourceware.org/gdb/current/onlinedocs/gdb/
2. Documentation de référence : http://valgrind.org/docs/manual/manual.html

2.3 Gestion des ressources

1. En entrant la commande suivante, exécutez votre programme corrigé sous valgrind.

```
valgrind --leak-check=full ./stack_ex2 ../Test/testfile1.txt
```

   Que pouvez-vous conclure sur l'utilisation des ressources de votre programme ?

2. Corrigez votre programme pour résoudre le problème identifié et vérifier la bonne correction en utilisant `valgrind`.