EPFL CS212 : travail de la semaine 03

J.-C. Chappelier, E. Bugnion, M. Sutherland & M. Stojilovic
EPFL

% Rev. 2021.03.05 / 7

Table des matières

Projet programmation système W03

Introduction

Cette semaine, nous voulons vous faire travailler les outils de débogage qui vous seront utiles pour la suite afin d'être plus efficaces dans la production de votre projet.

Le rendu de cette semaine comporte deux codes à déboguer : un petit, `stats.c`, et un plus conséquent, `muimp.c`.

Matériel fourni

Pour récupérez les fichiers fournis pour cette semaine, il vous faut rejoindre le troisième devoir de ce cours en suivant ce lien : https://classroom.github.com/a/E3i7-ICD.

Vous devriez alors avoir accès à un nouveau dépôt du genre `pps21-week03-YOURGITHUBID.git`, qu'il vous faudra donc cloner sur votre machine :

```
git clone git@github.com:projprogsys-epfl/pps21-week03-YOURGITHUBID.git
```

Comme d'habitude, ce dépôt contient un répertoire `done` qui contient deux programmes fournis, que vous devrez cette fois corriger.

Outils pour déboguer

Pour vous aider à trouver des fautes dans un code (en particulier dans votre propre code plus tard dans le projet ; pensez-y !), il existe plusieurs outils :

- les options du compilateur ;
- l'analyse statique de code ;
- l'analyse dynamique de mémoire (non présenté ici ; mais nous y reviendrons lorsque vous aurez des pointeurs dans votre projet (semaine 6)) ;
- et bien sûr les débogueurs.

Options utiles du compilateur

Le compilateur est d'une grande aide lorsque l'on sait bien l'utiliser et interpréter ses messages. Son comportement, plus ou moins verbeux, peut être modifié à l'aide d'options de compilation, dont nous vous détaillons les plus utiles ici.

Dans le même état d'esprit (utiliser le compilateur pour trouver des erreurs), il peut aussi être utile d'utiliser différents compilateurs (avec les options ci-dessous) sur le même code, car ils ne détectent pas forcément les mêmes choses. Sur les VMs, vous avez `gcc` et `clang`.

La première chose à faire est de bien spécifier la norme utilisée (car il existe de nombreux « dialectes » non normés). Cela se fait avec l'option `-std=`. Nous vous recommandons `-std=c99` ou `-std=c17`. Pour coller strictement à la norme spécifiée (et rejeter le « dialecte GNU » associé) ajoutez l'option `-pedantic`.

Ensuite, il peut être utile de laisser le compilateur nous avertir avec plein de *warning* usuels. Cela se fait avec l'option `-Wall` (comme « *all warnings*», même si en fait ils ne sont **pas** tous là `;-)`).

Pour avoir encore plus de warning, ajoutez `-Wextra`.

Et en voici encore d'autres que nous trouvons utiles d'ajouter (libre à vous de ne pas le faire si vous les trouvez trop pointilleux) :

- `-Wuninitialized` : avertit en cas de variable non initialisée ;
- `-Wfloat-equal` : avertit en cas de test d'égalité sur des nombres à virgule flottante ;
- `-Wshadow` : avertit si un nom en masque un autre (risque de problème de portée) ;
- `-Wbad-function-cast` : avertit en cas de mauvaise convertion de type de retour de fonctions ;

— `-Wcast-qual` : avertit en cas de conversion de type pointé qui supprime un qualificatif (comme `const` typiquement) ;
— `-Wcast-align` : avertit en cas de de conversion de type pointé qui ne respecte pas l'alignement des mots mémoire ;
— `-Wwrite-strings` : avertit en cas de (risque de) confusion entre `const char *` et `char *` ;
— `-Wconversion` : avertit en cas de conversion *implicite* de type ;
— `-Wunreachable-code` : avertit en cas de code inutile (non atteignable) ;
— `-Wformat=2` : augmente par rapport à `-Wall` le niveau d'avertissement au sujet des formats (genre `printf` et `scan`) ;
— `-Winit-self` : avertit en cas de d'initialisation récursive (genre `int i = 3 * i;`) ;
— `-Wstrict-prototypes` : avertit en cas de déclaration de fonction sans ses arguments ;
— `-Wmissing-declarations` : avertit en cas de fonction définie mais non prototypée ; cela peut être utile pour détecter l'oubli d'un prototype dans un `.h` (ou l'oubli d'un `#include`).

Enfin, vous pouvez bien sûr rajouter d'autres options si elles vous semblent utiles. Comme d'habitude, allez voir les « *man pages* » pour plus de détails.

Analyse statique de code

L'analyseur statique de code est un outil qui essaye de trouver des erreurs dans du code en « imaginant » tous les chemins d'exécution possibles. L'analyseur `scan-build` (et `scan-view`) est disponible sur les VMs. Il s'utilise en ajoutant simplement `scan-build` devant la commande de compilation, comme par exemples :

```
scan-build make
scan-build make cecicela
scan-build gcc -o monexo monexo.c
```

Le plus simple est d'essayer :

```
scan-build gcc stats.c -lm
```

Cette commande vous dit (tout à la fin) de regarder son analyse en utilisant `scan-view`, p.ex. :

```
scan-view /tmp/scan-build-2020-01-17-175346-107146-1
```

(mais ce dernier nom de fichier change à chaque fois).

Nous vous laissons regarder ce qu'elle a trouvé…

Utilisation d'un débogueur

Voir ce tutoriel pour l'utilisation du débogueur `gdb`.

Méthodologie pour débogueur

Pour trouver efficacement une erreur, nous vous proposons les conseils *généraux* suivants (d'autres conseils plus spécifiques au travail à faire sont également fournis plus bas) :

1. ne cherchez à corriger qu'un seul bug à la fois ;
2. commencez toujours par la première erreur ;
3. isolez/identifiez bien le bug de façon reproductible : re-testez toujours avec *exactement* les mêmes valeurs à chaque fois ;
4. appliquez la méthodologie suivante (cela peut sembler trivial, mais nous avons trop souvent vus des étudiants perdre leur temps à chercher des bugs au mauvais endroit car l'un des 2 « points » suivants n'était pas placé du bon coté ; souvent en raison d'hypothèses trop forte (suppositions fausses) ou de déductions fausses/trop rapides) :
   - ayez toujours 2 endroits (2 « points ») clairs dans votre code :
     - un endroit où vous êtes **absolument sûr** que le bug ne s'est pas encore produit (par exemple le tout début du programme) ;
     - et un autre où vous êtes **absolument sûr** que le bug s'est produit (par exemple l'endroit où le programme crashe, ou simplement la fin début du programme) ;
   - déplacez (avancez/remontez) le plus prometteur de ces points en étant sûr de ne pas « passer de l'autre coté » du bug ; vérifier cet aspect (« ne pas être passer de l'autre coté ») avec certitude ;
   - à la fin de ce processus (de recherche dichotomique en fait), les 2 « points » seront exactement sur l'endroit du bug.
5. si vous cherchez vos bugs à l'aide de messages affichés (`printf()`) :
   - mettez toujours un `\n` à la fin de chacun des messages ;
   - marquez le début de **chacun** de vos messages de débogage par un identifiant clair réservé uniquement à cela (p.ex. « `###` ») ; cela vous permet :
     1. de voir facilement ces messages dans la sortie du programme ;
     2. de les retrouver ensuite facilement dans votre code pour les éditer/supprimer plus tard ;
   - ayez dans **chaque** message une partie qui lui est unique (p.ex. « debug(1): », « debug(2): », « debug(3): », etc., ou encore « ici i= », « ici j= », « ici k= », etc. ; vous pouvez bien sûr combiner) ;

   avoir cette discipline avec ses messages de débogage peut sembler une perte de temps (surtout quand on est en train de chercher le bug), mais, croyez moi, cela *sauve* en fait *beaucoup* de temps au final !

Travail à rendre : cas simple

Tous les outils précédents vous seront utiles pour être plus efficaces dans votre projet. Nous vous demandons donc de commencer à les pratiquer pour corriger

le code `stats.c` fourni.

Le premier rendu pour cette semaine (délai : dimanche 21 mars 23:59) consiste en la version *entièrement* corrigée de `stats.c` dont le but est de calculer la moyenne et l'écart-type (non biaisé) de l'âge d'un ensemble de 1 à 1024 personnes (attention ! il contient *plusieurs* erreurs, de différente nature ; il n'y a par contre pas d'erreur de nature mathématique : les *formules* sont correctes du point de vue mathématique ; mais notez cependant que l'écart-type d'une population réduite à un seul individu doit être zéro).

Le code rendu doit être correct et robuste à toute entrée de type entier (mais nous **ne** vous demandons **pas** de traiter le cas d'entrées non entières composées de caractères quelconques ; nous ne testerons votre code qu'avec des entiers, positifs, négatifs ou nuls).

Le rendu se fera automatiquement le dimanche 21 mars à 23:59 en prenant la version enregistrée (`commit` + `push`) dans le répertoire `done` votre dépôt GitHub (branche `master`). Il est donc impératif que vous ayez `commit` puis `push` votre version finale avant cette date. Aucune autre solution ne sera acceptée.

Travail à rendre : second cas, plus complexe

Le deuxième rendu pour cette semaine (délai : dimanche 21 mars 23:59, aussi) est la version *entièrement* corrigée de code `muimp.c` fourni (quand nous disons «*entièrement*», c'est du point de vue d'un enseignant de programmation, tous les aspects, bugs, mais aussi toute mauvaise pratique, doivent être corrigés ; et il y en a beaucoup, de tous ordres). Ce code, une fois les erreurs corrigées, devrait permettre de créer une image numérique simplifiée, l'afficher (en mode texte), l'écrire et la lire depuis un fichier ainsi que de lui appliquer un filtrage par convolution. La description détaillée de la structure et des fonctionnalités attendues de ce code se trouve ici (et ici en PDF, plus lisible pour les maths si nécessaire). Il n'est pas forcément nécessaire de commencer par lire cette description car le code `muimp.c` fourni contient tellement d'erreurs (de différentes natures) que commencer par lire le code lui-même, le compiler avec des options, etc. devrait déjà permettre de corriger plusieurs choses. Lisez le descriptif en question que pour trouver des erreurs plus subtiles ou en cas de doute lors de la lecture du code fourni. C'est toujours le descriptif qui fait foi (jamais le programme fourni).

Comme pour l'exercice précédent, le rendu se fera automatiquement le dimanche 21 mars à 23:59 en prenant la version enregistrée (`commit` + `push`) dans le répertoire `done` de votre dépôt GitHub (branche `master`). Il est donc impératif que vous ayez `commit` puis `push` votre version finale avant cette date. Aucune autre solution ne sera acceptée.

Tests

Bien sûr, pour corriger des programmes il faut aussi les tester pendant leur exécution !!

Pensez donc à tester vos programmes au delà des exemples de déroulement éventuellement fournis (cela fait aussi partie des objectifs de ce devoir et devrait être – sinon devenir – un réflexe systématique).
Imaginez non seulement les cas standards, mais aussi **tous** les cas particuliers.

Comme d'habitude, nous fournissons également, *à bien plaire*, un `make feedback` (`make feedback-VM-CO` si vous travaillez sur les VM de l'Ecole) qui donne un retour *partiel* sur votre travail.

Voici également quelques pistes spécifiques à la correction des deux programmes demandés :

`stats`

La première chose à faire serait peut être de compléter le `Makefile` pour qu'il puisse produire `stats` avec les informations utiles au débogueur (option `-g`). Vous pourriez aussi en profiter pour allumer les *warnings* du compilateur et regarder en détail ce qu'il vous dit et surtout comprendre *pourquoi* il vous le dit.

Une fois que le programme compile, si possible sans *warning*, voici 3 pistes pour aller plus loin dans la correction du programme: 1. testez des valeurs qui sont *hors* des limites attendues, et regardez si le programme réagit comme vous attendriez. Par exemple : que se passe-t-il si on entre un nombre négatif de personnes ? un age négatif ? 2. calculez à la main la moyenne et l'écart-type (suivant la formule *donnée* !) d'un petit échantillon et comparez les à la sorite de votre programme. S'il y a des différence, utilisez le débogueur pour chercher d'où elles viennent ; 3. pensez à tester *tous* les cas limites de ces formules.

`muimp`

Dans ce programme, il n'y a pas que des erreurs fonctionnelles, mais aussi beaucoup d'erreurs de style, de « bonnes pratiques » ou même de logique (p.ex. vous semble-t-il logique de commencer à demander la diagonale si celle-ci doit être inférieure à la largeur et la hauteur ?). Autre exemple, vous devriez déjà savoir qu'il n'est pas bon d'avoir des « *magic numbers* » dans un programme.
Les 2 tests fourni dans `make feedback` ne testent **que** les fonctionnalités, pas le reste (de plus, ces tests attendent un format plus ou moins précis pour les entrées sorties, en particulier l'ordre des arguments est : largeur, hauteur, diagonale). Il est donc nécessaire que vous prenniez du recul par rapport à ce feedback.
(Et, encore une fois, ceci est normal : vous n'aurez jamais, dans la vraie vie, la vraie réponse absolue (si tant est qu'elle existe !) ; c'est bien une composante à par entière du métier d'ingénieur informaticien que de prévoir les cas de tests.)

Après avoir lu le code et corrigé les erreurs les plus évidentes, y compris celles

détectées par le compilateurs, ses *warnings* et l'analyse statique, nous vous suggérons de continuer comme pour `stats` en essayant par exemple des valeurs hors des bornes attendues. Après cela, vous pouvez envisager de regarder des cas plus complexes comme les deux testés par `make feedback`.

Ces deux scénarios testés par `make feedback`, ils sont conçus pour mettre le doigt sur 2 erreurs dans la logique du code. Là encore, ce serait à notre avis plus efficace d'utiliser le débogueur comme expliqué plus haut. Afin de vous aider à suivre ces deux scénarios, `make feedback` affiche les paramètres qu'il utilise pour tester `muimp`.

Le premier de ces deux tests, « *feedback_diag* » vérifique que le code produit une « image » rectangle de la bonne taille et le losange correspondant. Utilisez les paramètres indiqués par `make feedback` pour déboguer le code, si nécessaire.
Le second test, « *feedback_str_filter* » teste la possibitilté d'avoir de long noms de fichiers, ainsi que le fait que `filter()` fonctionne correctement.

Happy debugging! All bugs are puzzles that need to be solved. :)

En espérant que cette semaine vous apportera les moyens d'être plus efficace dans votre vie de développeuse/développeur...

Rendu

Pour rendre le devoir, « commitez » (`commit`) et « poussez » (`push`) toutes vos modifications (fichiers `stats.c` et `muimp.c`) avant le dimanche 21 mars 23:59. Pensez également à mettre votre fichier `time.csv` tel que décrit dans le barème du cours. Le rendu se fera automatiquement en prenant la version enregistrée (`commit` + `push`) dans le répertoire `done` de votre dépôt GitHub (branche `master`) à cette date. Il est donc impératif que vous ayez `commit` puis `push` votre version finale avant cette date. Aucune autre solution ne sera acceptée.

Et n'oubliez pas de faire le rendu de la semaine passée avant dimanche soir (de cette semaine).