

# TP4: Bug hunting – Correction

## Module Sys1

### Objectifs pédagogiques:

- Comprendre les limitations du debug sans outil;
- Apprendre à utiliser les outils valgrind et gdb pour corriger des bugs;
- Savoir utiliser des outils pour découvrir de nouveaux bugs.

À faire avant la séance : Installer les dépendances et vérifier que le code fourni compile correctement.

L'objectif de ce TP est de vous donner des pistes sur comment trouver les bugs dans vos programmes.

La recherche d'une erreur au sein d'un programme est une sorte de jeu de pistes où l'on recherche des informations sur le contexte, les symptômes, les causes possibles de l'erreur. Cela permet de déterminer sa localisation et la manière de la corriger. La méthode traditionnelle consistant à utiliser la commande `printf` en divers endroits du programme est l'expression de cette recherche d'information. Des outils tels que `gdb` et `valgrind` facilitent l'obtention d'informations sur les programmes.

### Réponse

La correction de ce TP n'apporte pas forcément grand chose car le sujet est déjà extrêmement rédigé et un peu directeur. Pour l'instant, il contient surtout des spoils des lignes contenant les bugs que l'on cherche. Si vous avez des idées de choses à mieux expliquer dans la correction, dites moi.

### Fin réponse

★ **Code fourni.** Ce TP se présente sous la forme d'un programme complet (jeu de Sokoban) dans lequel 7 bugs ont été ajoutés. Le code est en ligne : <https://mquinson.frama.io/ensr-arcsys1/TP4-bug-hunting.tgz>

Pour compiler le programme, il est tout d'abord nécessaire d'installer les dépendances avec la commande `apt-get install libsdl2-dev build-essential valgrind` ou `pacman -S sdl2 make valgrind`. La compilation peut ensuite être lancée avec la commande `make`, qui produira l'exécutable `./sokoban`.

★ **Exercice 1: la méthode printf (1 bug).**

Cette méthode est utilisée dans les cas où on ne peut (ou ne veut) pas utiliser de debugger. Attention cependant au piège classique de cette méthode, mis en valeur par le premier bug.

Exécuter le programme devrait afficher ce qu'on voit ci-dessous à gauche. En observant le fichier source `sok.c` au début de la fonction `main`, autour de la ligne 1500 (ci-dessous à droite), on peut en déduire que le problème se trouve entre la ligne 1495 (qui s'affiche bien) et la ligne 1506 (qui ne s'affiche pas).

```
$ ./sokoban
1/ SDL initialized
2/ Window created
Erreur de segmentation
$
```

```
1491 if (window == NULL) {
1492     printf("Window could not be created! SDL_Error: %s\n", SDL_GetError());
1493     return(1);
1494 } else {
1495     printf("2. Window created\n");
1496 }
1497 setsokicon(window);
1498 SDL_SetWindowMinimumSize(window, 160, 120);
1499
1500 renderer = SDL_CreateRenderer(window, -1, 0);
1501 if (renderer == NULL) {
1502     SDL_DestroyWindow(window);
1503     printf("Renderer could not be created! SDL_Error: %s\n", SDL_GetError());
1504     return(1);
1505 } else {
1506     printf("3. Renderer created");
1507 }
```

Pourtant, le problème est localisé juste en dessous, à la ligne 1510.

▷ **Question 1:** Pourquoi cette ligne provoque-t-elle une erreur ?

**Réponse**

Parce que le pointeur `levelfile` a encore sa valeur initiale `NULL` attribuée en ligne 1459. Déreferencer le pointeur nul mène dans le mur comme il se doit.

**Fin réponse**

Le problème a lieu après la ligne 1506, et pourtant cet affichage n'a pas lieu. C'est parce que les affichages de `printf` ne sont pas toujours réalisés immédiatement. Pour des raisons de performances, `printf` minimise le nombre d'actions d'affichage, quitte à afficher plus de texte à chaque fois. Pour cela, le texte passé à `printf` est parfois placé dans un tampon pour être affiché plus tard. Malheureusement, si le programme est tué brutalement, ces messages ne seront jamais affichés.

```

bug-boom.c
#include <stdio.h>
1
2
int main() {
3
4
5
6
7
8
9
10
11
12
13
    int *p;

    printf("1");
    p = NULL;
    printf("2");
    *p = 1;
    printf("3");

    return 0;
}

```

▷ **Question 2:** Que pensez-vous que le programme `bug-boom.c` ci-contre affiche? Vérifiez en compilant le source, également fourni dans l'archive.

Les `printf` suggèrent donc une localisation erronée du problème, ce qui peut faire perdre un temps considérable. Plusieurs solutions permettent d'éviter ou au moins de contrôler cette mise en tampon.

▷ **Question 3:** Quel est l'affichage si vous ajoutez retour-chariot en ligne 6?

```
printf("1\n");
```

Et si vous lancez votre programme dans un tube : `./boom|less` ?

**Réponse**

C'est parce que le système vide le tampon à chaque fin de ligne, mais seulement l'affichage est dirigé sur un terminal. C'est différent si c'est dans un tube. Les concepteurs du système ont dû se dire qu'un affichage destiné à un programme est moins urgent et qu'il n'est pas nécessaire de vider le tampon dans ce cas.

**Fin réponse**

▷ **Question 4:** Changez la ligne 8 pour utiliser `fprintf` sur le canal `stderr`:

```
fprintf(stderr, "2");
```

Quel est maintenant le comportement de votre programme? Et si la sortie est un tube?

**Réponse**

C'est parce que la sortie d'erreur n'est pas mise en tampon, car les messages d'erreurs sont considérés urgents et doivent être affichés au plus vite, même si cela induit une perte de performances.

**Fin réponse**

▷ **Question 5:** Ajoutez un `fflush(stdout)` après la ligne 6. Quel est maintenant le comportement de votre programme? Et si la sortie n'est pas un terminal mais un tube?

**Réponse**

C'est parce que la fonction `fflush` a pour objectif de pour vider le tampon et forcer l'affichage immédiat des informations. Cette question n'a pas d'autre objectif que de présenter cette fonction, en fait.

**Fin réponse**

▷ **Question 6:** Résumez dans ce tableau si l'affichage a lieu immédiatement ou non dans chacun des cas.

	<code>printf("aze")</code>	<code>printf("aze\n")</code>	<code>fprintf(stderr, ...)</code>	<code>printf()+fflush()</code>
Sortie sur un terminal				
Sortie sur un tube				

**Conclusion.** Cet exercice nous a permis d'explorer le principal piège de la mise au point à base de `printf`, qui peut suggérer une localisation erronée du problème. Nous avons vu trois façons de contourner ce piège, mais cette méthode reste artisanale. Il est souvent nécessaire d'utiliser des outils spécialisés comme `valgrind` et `gdb`.

### ★ Exercice 2: La suite d'outils `valgrind` (2 bugs).

Lancez le programme `simplesok` après avoir corrigé le problème de la ligne `sok.c:1510`. Un nouveau message d'erreur devrait apparaître dès le lancement, comme indiqué ci-contre.

Nous allons utiliser `valgrind` pour localiser la source de cette nouvelle erreur. Il s'agit d'une suite d'outil fabuleuse pour mettre au point vos programmes. Selon l'outil utilisé, il est possible de détecter la plupart des

```

$ ./simplesok
1/ SDL initialized
2/ Window created
3/ Renderer created
free(): double free detected
Abandon (core dumped)
$

```

problèmes liés à la mémoire (outil `memcheck`), d'étudier les effets de cache pour améliorer les performances (avec `cachegrind`), de déboguer des programmes multi-threadés (avec `hellgrind`) ou encore d'optimiser les programmes (avec `callgrind`).

▷ **Question 1:** Ce programme fonctionne mieux si on demande au compilateur d'inclure des informations de debug aux programmes générés. Il faut ajouter l'option `-g` à la ligne de compilation. Modifiez le `Makefile` pour ajouter cette option à la fin de la première ligne. La variable `CFLAGS` contient les options à passer au compilateur.

▷ **Question 2:** Lancez votre programme dans `valgrind`: `valgrind --tool=memcheck ./simplesok`

`Valgrind` ajoute alors de nombreuses lignes commençant par `==<identifiant du processus>==` à l'affichage normal du programme.

Le problème de `valgrind` est qu'il est vraiment efficace, et il est possible que vos affichages soient pollué par des problèmes localisés dans la bibliothèque `SDL`. Il est possible de lister à `valgrind` des erreurs à ignorer (avec un fichier de suppression), mais vous pouvez vous contenter d'attraper la sortie de `valgrind` dans un `less` pour pouvoir la lire à votre rythme: `valgrind --tool=memcheck ./simplesok 2>&1 | less`

▷ **Question 3:** Utilisez les informations fournies pour localiser la source de ce nouveau problème, et résolvez-le.

**Réponse**

BUG#2: ligne 592, supprimer `free(string);`.

**Fin réponse**

▷ **Question 4:** Une fois ce second bug résolu, vous pouvez accéder au menu, mais un troisième bug apparaît dès que l'on sélectionne l'une des entrées. Utilisez de nouveau `valgrind` pour localiser et corriger le problème.

Sur mon ordinateur, je dois ignorer deux erreurs de type `Invalid read` localisées dans `SDL` pour trouver un problème localisé dans le fichier `sok.c`. Vous pouvez ensuite reformater ou même réécrire un peu le programme source pour décomposer les opérations et comprendre où se trouve le problème.

**Réponse**

BUG#3: ligne 1692, double `==` dans la condition.

**Fin réponse**

Un quatrième bug apparaît lorsque l'on sélectionne l'entrée "Easy" du menu, mais nous allons devoir passer à l'étape supérieure et dégainer `gdb` pour le localiser.

### ★ Exercice 3: Utilisation de base du debugger GNU: `gdb` (1 bug).

▷ **Lancement de `gdb`.** Tapez la commande `gdb ./simplesok` pour charger votre programme dans l'environnement `GDB`. On contrôle ce programme en tapant des commandes à l'invite. Les commandes les plus importantes sont `help`, `list`, `quit` et `run`.

▷ **Question 1:** Essayez la session suivante dans `gdb` :

- Chargez `simplesok` dans `gdb` et lancez le programme avec la commande `run` dans la console `gdb`.
- Tapez `<ctrl+c>` pour interrompre votre programme.
- Visualisez le code en cours d'exécution avec la commande `list`.
- Reprenez l'exécution avec `cont`, puis interrompez-la de nouveau. Que constatez-vous ?

▷ **Question 2: Interface "graphique" de `gdb`.** Refaites la session précédente dans `gdb`, mais après l'avoir lancé de la façon suivante: `gdb -tui ./simplesok` On peut activer cette *Terminal User Interface* à tout moment dans `gdb` avec la séquence de touches `Ctrl-x a`

▷ **Question 3:** À l'aide des commandes `print v` (qui affiche le contenu d'une variable `v`) et `jump n` (qui fait sauter l'exécution à la ligne `n` — oui, cela modifie à chaud l'exécution du programme), expliquez la cause du problème et résolvez-le.

**Réponse**

BUG#4: ligne 1108, déclarer `maxallowedlevel` en `int`.

**Fin réponse**

### ★ Exercice 4: Trois derniers (?) bugs dans Simple Sokoban.

Cette partie n'est pas guidée. Vous avez résolu suffisamment de bugs pour lancer une partie, mais trois autres bugs devraient continuer à apparaître. Utilisez `valgrind` et `gdb` pour résoudre ces problèmes. Si vous trouvez plus que 7 bugs au total, c'est que le jeu utilisé pour construire ce TP était buggé avant notre intervention :)

#### Réponse

- BUG#5: `sok_core.c`, ligne 424, `states->historyallocsize *= 2`; sinon le `realloc` juste après n'agrandit rien.
- BUG#6: `sok_core.c`, supprimer lignes 392-394.
- BUG#7: `save.c`, supprimer ligne 170.

#### Fin réponse

### ★ Exercice 5: Du bon usage du compilateur pour trouver des problèmes.

Certaines options demandent au compilateur de signaler les problèmes potentiels qu'il détecte. Il est recommandé de toujours activé au **minimum** ces deux options : `-Wall -Wextra` D'autres options sont utiles:

```
-Wwrite-strings -Wcast-align -Wformat-security
-Wformat-nonliteral -Wpointer-arith
-Wmissing-include-dirs -Wmissing-declarations -Wundef
-Wmissing-prototypes -Wformat=2 -Wsign-conversion
-Wunused-macros -Wswitch-bool -Wredundant-decls
-Wlogical-op -Wdouble-promotion -Wbool-compare
-Wlogical-not-parentheses
```

Leur signification, ainsi que la raison pour laquelle ces options ne sont pas ajoutées par défaut dans `-Wall` se trouve dans la documentation de `gcc`, en ligne.

- ▷ **Question 1:** Saurez-vous trouver les trois bugs du programme ci-contre en le lisant?
- ▷ **Question 2:** Compilez `bug-boucle.c` en activant des warnings, et corrigez les problèmes rapportés.
- ▷ **Question 3:** Corrigez les derniers problèmes détectés par `valgrind`.

```

bug-boucle.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int *tab = NULL;
5
6 void initialise(int n)
7 {
8     char i = 0;
9
10    for (i = 0; i <= n; i++);
11    {
12        tab[i] = 1;
13    }
14 }
15
16 int main()
17 {
18    printf("Debut\n");
19    tab = malloc(10000*sizeof(int));
20    initialise(10000);
21    printf("Fin\n");
22
23    return 0;
24 }
```

### ★ Exercice 6: Usage plus avancé de gdb (optionnel)

Nous allons maintenant utiliser le debugger avec un autre programme afin d'expérimenter les opérations permettant de trouver les problèmes impliquant des fonctions. Chargez `bug-fact.c` dans `gdb` après compilation.

#### Points d'arrêt et exécution pas à pas

Lors de la traque d'une erreur, il est fréquent d'avoir une idée de sa localisation potentielle. `gdb` permet donc de spécifier des points d'arrêt dans le code où l'exécution est automatiquement interrompue. La commande `break` suivie d'un nom de fonction ou d'un numéro de ligne (éventuellement associé à un fichier) insère un point d'arrêt à l'endroit spécifié. `clear` supprime le point d'arrêt spécifié.

Placez un point d'arrêt sur la fonction `main` puis lancez l'exécution. Elle s'interrompt avant le début du code. Expérimentez avec les commandes `next` et `step`. Chacune permet d'avancer l'exécution d'une ligne puis de bloquer l'exécution. Si cette ligne contient un appel de fonction, `step` entre dans le code de cette fonction tandis que `next` l'exécute en entier et passe à la ligne suivante de la fonction courante.

**Pile et cadres** La commande `backtrace` permet d'afficher la pile d'exécution du processus. Spécifiez un point d'arrêt sur la ligne 9 (`x=1`) et lancez l'exécution. Lorsque le processus est stoppé, exécutez `backtrace`.

La liste affichée indique tout d'abord les appels récursifs à `fact` et termine par `main`. Les fonctions sont donc listées depuis l'appel le plus imbriqué (regardez la valeur indiquée pour le paramètre `n` de `f` pour chaque cadre) vers l'appel le moins imbriqué (donc dans l'ordre inverse de l'ordre chronologique, d'où le nom de la commande).

Chaque ligne constitue ce que l'on appelle un *cadre de pile* (« `frame` » en anglais). Il est possible de se déplacer dans la pile avec les commandes `up` et `down`, ou directement avec la commande `frame` suivie du numéro de cadre visé.

**Affichage de variables et d'expressions** La commande `print` permet d'afficher le contenu d'une variable. Placez un point d'arrêt sur `fact` puis ré-exécutez. Utilisez `print n`. La commande `disp` est similaire, mais affiche le résultat à chaque interruption du programme. Exécutez `disp (char)n+65` une seule fois, puis utilisez `cont` plusieurs fois.

On peut de plus modifier des valeurs avec `set variable VAR=EXP` où `VAR` est le nom de la variable à modifier et `EXP` l'expression dont le résultat est à lui affecter. Si le nom de la variable à modifier n'entre pas en conflit avec les variables internes de `gdb`, on peut omettre le mot-clé `variable`.

**Conclusion sur gdb.** Vous en savez maintenant assez sur `gdb` pour faire vos premiers pas. Il existe cependant de nombreuses fonctionnalités que nous n'avons pas abordé ici comme les *watchpoints* (qui arrêtent l'exécution quand une variable donnée est modifiée), le chargement de fichiers *core* pour débogger un programme après sa mort, la prise de contrôle de processus en cours d'exécution, et bien d'autres encore. `info gdb` pour les détails.

```

bug-fact.c
#include <stdio.h>
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
int fact(int n) {
    int x = 0;
    if (n > 0) {
        x = n * fact(n - 1);
    } else {
        x = 1;
    }
    return x;
}

int main() {
    int a = 10;
    int b = 0;
    b = fact(a);
    printf("%d!=%d\n", a,b);
    return 0;
}

```

★ **Pour aller encore plus loin (pour information).**

`Valgrind` n'est pas le seul outil dans la catégorie des **analyseurs dynamiques**, basés sur une exécution du code après modification. Le fonctionnement de `valgrind`, détaillé dans les articles scientifiques correspondants, revient à décompiler le binaire, insérer des vérifications et gardes-fous dans le programme, et recompiler le tout. Comme `valgrind` n'est pas un vrai compilateur, on perd beaucoup d'efficacité lors de la recompilation.

Les *sanitizers* de `clang` sont des outils intégrés à la suite de compilateurs LLVM et celle de `gcc`. Leur principe est le même que `valgrind`, mais ils instrumentent le code (ils ajoutent des instructions supplémentaires) au moment de la compilation classique. Il faut donc recompiler son programme en ajoutant un paramètre `-fsanitize=??` à la ligne de compilation (on l'ajoute aux `CFLAGS` comme on a ajouté `-g` précédemment). En échange de cette complexité accrue, le programme ainsi compilé va bien plus vite qu'avec `valgrind`. Les *sanitizers* principaux peuvent détecter à l'exécution des comportements indéfinis (`-fsanitize=undefined`), une mauvaise utilisation de la mémoire ou encore l'oubli de libérer de la mémoire allouée dynamiquement (`-fsanitize=address`). De nouvelles vérifications sont ajoutées au fil des versions, car ce projet intègre les dernières trouvailles de la recherche dans le domaine.

Dans la catégorie des **analyseurs statiques** aussi, on trouve de nombreux outils en plus des options de warning des compilateurs. La plupart d'entre eux sont relativement lents, ce qui explique qu'ils ne sont pas inclus directement dans les compilateurs pour ne pas ralentir la génération des binaires. Le meilleur projet libre existant dans cette catégorie est certainement `clang-analyzer`, de la famille LLVM. `clang-tidy` est un autre outil de cette famille, qui vise plus à faire des vérifications stylistiques qu'à trouver des bugs complexes.

`SonarQube` et `LGTM` sont d'autres outils d'analyse statique de code, redoutablement efficaces pour améliorer la qualité d'un gros projet. Ils ne s'agit malheureusement pas de logiciels libres, même si leur usage est gratuit sous conditions. `Infer` est un autre très bon outil d'analyse de code, qui a en plus le bon goût d'être open source.

Tous ces outils peuvent aider à améliorer la qualité de vos projets (scolaires ou autre), et vous économiser des heures de debug. Plus encore, ils constituent une application pratique de recherches assez intéressantes.