TP5 : Makefiles et mise au point de programmes C

En plus de la compilation séparée et l'utilisation de Makefile, 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 lqa manière de la corriger. La méthode traditionnelle consistent à utiliser la commande `printf` en diverhttp ://mobile.free.fr/s 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.

★ Exercice 1. Compilation séparée et Makefile.

Récupérez le code fourni dans le dépot.

▷ **Question 1.** Vérifiez que les commandes `gcc -c code1.c` et `gcc -c code2.c` ne produisent pas de message d'erreur et que les fichiers ***code1.o*** et ***code2.o*** ont été créés.

▷ **Question 2.** À partir de ces deux fichiers et de ***codage.c***, créez le fichier exécutable ***codage*** par la commande `gcc -o codage codage.c code1.o code2,o`

▷ **Question 3.** Exécutez-le, puis supprimez les fichiers ***code1.o***, ***code2.o*** et ***codage***.

▷ **Question 4.** Écrivez un fichier Makefile permettant de compiler ce projet. On rappelle qu'un tel fichier est composé de différentes règles, chacune suivant la syntaxe suivante (avec un caractère tabulation et non des espaces au début des lignes de commande) :

```
<fichier cible> : <liste des dependances>
<commande pour fabriquer la cible a partir des dependances>
```

▷ **Question 5.** Vérifiez que votre makefile exécute les bonnes actions dans les cas suivants :
- Aucun binaire n'existe, tout est à reconstruire ;
- Effacement de l'un des `.o` ;
- Effacement du binaire ;
- Modification de l'un des fichiers source ;

★ Exercice 2. Mise au point : la méthode printf

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 dans le programme `boom.c` ci-contre (également dans le dépot).
Ce programme devrait afficher `12Erreur de segmentation` puisque la ligne 9 revient à déréférencer le pointeur NULL, ce qui est interdit.

Listing 1 – boom.c

```
1  #include <stdio.h>
2
3  int main() {
4      int *p;
5
6      printf("1");
7      p = NULL;
8      printf("2");
9      *p = 1;
10     printf("3");
11
12     return 0;
13 }
```

▷ **Question 1.** Quel est l'affichage généré par ce programme ?
C'est parce que les affichages de printf ne sont pas toujours réalisées immédiatement. Pour des raisons de performances, le système cherche en effet à retarder les affichages de façon à avoir moins d'action d'affichage pour plus de texte à chaque fois. C'est pourquoi les "1" et "2" sont placés dans un tampon pour être affichés plus tard. Malheureusement, comme l'erreur de segmentation de la ligne 9 tue brutalement le programme, ces messages ne seront jamais affichés.

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 2.** Ajoutez des retours-chariots à la fin des affichages (la ligne 6 devient `printf("1\n");`). Quel est maintenant l'affichage de votre programme ? Et si vous lancez votre programme de la façon suivante : `./boom|less` ?

C'est parce que le système vide le tampon à chaque fin de ligne si et seulement l'affichage est dirigé sur un terminal.

▷ **Question 3.** Retirez les \n que vous aviez ajouté à la question précédente, et demandez à réaliser les affichages sur la sortie d'erreur (en utilisant `fprintf(stderr, "...")` à la place de printf. Quel est maintenant le comportement de votre programme ? Et si la sortie n'est pas un terminal mais un tube ?

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 petite perte de performances.

▷ **Question 4.** Rechangez vos affichages pour utiliser la sortie standard (avec `printf`), et ajoutez des `fflush(stdout)` après chaque `printf`. Quel est maintenant le comportement de votre programme ? Et si la sortie n'est pas un terminal mais un tube ?

C'est parce que la fonction `fflush` a pour objectif de pour vider le tampon et forcer l'affichage immédiat des informations.

Conclusion. Cet exercice nous a permis d'explorer le principal piège de la mise au point à base de `printf`. Nous avons vu 3 façons de contourner ce piège, mais cette méthode reste artisanale, et il est souvent nécessaire d'utiliser des outils spécialisés comme `gdb`.

★ Exercice 3. Le debugger GNU gdb - utilisation de base

Nous utiliserons comme premier exemple le programme `boucle.c` ci-dessous (également dans le dépot).

Pour le compiler, il convient d'utiliser la commande `gcc -Wall -g -o boucle boucle.c`. L'option `-Wall` demande l'activation de nombreux *warnings* (mais pas tous !) tandis que `-g` ajoute au binaire produit les informations de deboguage nécessaire à `gdb` (et autres debuggers).

Listing 2 – boucle.c

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

▷ **Question 1.** Exécutez ce programme. Que constatez vous ?

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

▷ **Question 2.** Essayez la session suivante dans gdb :

- Chargez boucle dans gdb et lancez le programme.
- Tapez <ctrl+c> pour interrompre votre programme.
- Visualisez le code en cours d'exécution avec `list`.
- Reprenez l'exécution avec `cont`, puis interrompez-la de nouveau. L'exécution n'a pas progressé.
- Aidez le programme à franchir la zone difficile à l'aide de la commande `jump 11`, ce qui fait sauter l'exécution à la ligne 11 (oui, cela modifie le schéma d'exécution du programme). Le programme doit se terminer normalement. Reste à comprendre pourquoi le programme ne passe pas la ligne 10 seul.

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.

▷ **Question 3.** Pour trouver le problème, interrompez au besoin votre programme (ctrl-C), utilisez la commande `print` pour afficher le contenu de la variable i (`print i`). Vous pouvez également le faire continuer (commande `continue`), et le réinterrompre. Corrigez le problème.

Indice : ce premier bug se trouve ligne 7.

▷ **Question 4.** Maintenant que le programme s'exécute jusqu'à la fin, on constate que l'affichage de la ligne 20 indique que l'affectation du tableau ne s'effectue pas correctement, puisque les cases valent 0 au lieu du 1 attendu. Réexécutez votre programme pas à pas pour comprendre le problème, puis corrigez le.

Indice : ce second bug se trouve ligne 9.

★ Exercice 4. Le debugger GNU gdb - utilisation avec les fonctions

Nous allons maintenant utiliser le debugger avec un autre programme afin d'expérimenter les opérations permettant de trouver les problèmes impliquant des fonctions.

Pile et cadres La commande `backtrace` permet d'afficher la pile d'exécution du processus. Compilez ***fact.c*** (page suivante et dans le dépot) puis chargez fact dans `gdb`. 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 étranger). 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` 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`, la prise de contrôle de processus en cours d'exécution, et bien d'autres encore. `info gdb` pour les détails.

Listing 3 – fact.c

```
1  #include <stdio.h>
2
3  int fact(int n) {
4      int x = 0;
5
6      if (n > 0) {
7          x = n * fact(n - 1);
8      } else {
9          x = 1;
10     }
11
12     return x;
13 }
14
15 int main() {
16     int a = 10;
17     int b =  0;
18
19     b = fact(a);
20     printf("%d!=%d\n", a,b);
21
22     return 0;
23 }
```

★ Exercice 5. La suite d'outils valgrind

`valgrind` est une suite d'outil fabuleuse pour mettre au point vos programmes. Selon l'outil utilisé, il est possible de détecter la plupart des problèmes liés à la mémoire (outil `memcheck`), d'étudier les effets de cache pour améliorer les performances (avec `cachegrind`), de débugger des programmes multi-threadés (avec `hellgrind`, voir le cours de système en 2A) ou encore d'optimiser les programmes (avec `callgrind`). Nous allons nous intéresser au premier outil, que l'on lance avec `valgrind --tool=memcheck <prog>`

▷ **Question 1.** Lancez valgrind sur le programme `boom` étudié plus tôt. S'affichent de nombreuses lignes commençant par `==<identifiant du processus>==`. Elles sont le fait de `valgrind`.

La cause de l'erreur de segmentation est donnée par le second groupe de ligne :

```
==29579== Invalid write of size 4
==29579==    at 0x80483CA: main (boom.c:9)
==29579==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

À la ligne `boom.c:9`, nous écrivons 4 octets (sans doute un entier) à un endroit invalide. En effet, l'adresse `0x0` [où nous tentons d'écrire] n'est ni sur la pile, ni le résultat d'un `malloc` et il n'a pas été free()é récemment. Bien sûr ! La ligne 9 écrit à l'adresse pointé par `p`, mais `p` vaut la valeur `NULL`, qui n'est pas une adresse valide (et on a `NULL=0x0`). `valgrind` localise immédiatement et précisément le problème.

▷ **Question 2.** Lancez maintenant `valgrind` sur le programme `boucle` (après avoir corrigé les deux bugs identifiés dans l'exercice 2). Vous pouvez constater que le programme que l'on croyait corrigé contient encore des problèmes :

```
==10816== Invalid write of size 4
==10816==    at 0x8048429: initialise (boucle.c:11)
==10816==    by 0x8048476: main (boucle.c:18)
==10816==  Address 0x41a7c68 is 0 bytes after a block of size 40,000 alloc'd
==10816==    at 0x402601E: malloc (vg_replace_malloc.c:207)
==10816==    by 0x8048465: main (boucle.c:17)
```

La ligne `boucle.c:11` tente d'écrire 4 octets à un endroit invalide. De plus, cet endroit est localisé juste après un gros bloc mémoire alloué en `boucle.c:17`. Corrigez ce problème (indice : le bug est en ligne 9).

▷ **Question 3.** Relancez `valgrind` sur le programme `boucle`. À la fin de l'exécution, valgrind affiche :

Il y a donc un bloc de mémoire (de 400 octets) obtenu par `malloc`, mais jamais restitué au système avec `free`. Ajoutez l'option nécessaire pour voir lequel et corrigez le problème.

★ Exercice 6. Organisation mémoire d'un processus

On souhaite explorer la disposition mémoire d'un processus C, c'est-à-dire les adresses mémoire où sont placés les différents éléments possible. Pour afficher l'adresse à laquelle se trouve une variable `i` quelconque, il faut utiliser : `printf("variable i: %p",&i);`

▷ **Question 1.**

- Dans un fichier appelé exo1.c, déclarer les variables suivantes :
  - chaîne de 10 caractères globale et statique ;
  - tableau de 10 entiers global ;
  - flottant local au main ;
  - caractère local statique ;
  - entier nommé `ex` déclaré en externe.
- Dans la fonction main(), faire écrire les adresses des variables.
- Dans un deuxième fichier appelé exo1bis.c mettre la définition de l'entier `ex` que vous initialiserez à la valeur 20.

▷ **Question 2.** Compilez ce programme avec `gcc -c exo1.c` `gcc -c exo1bis.c` puis réalisez l'édition de liens avec `gcc -o exo1 exo1.o exo1bis.o`. Exécutez `exo1` et concluez sur la place des différentes variables. Quelle est l'effet du modificateur `static` ? Qu'en est-il de `extern` ?

▷ **Question 3.** Modifiez votre programme pour qu'il affiche l'adresse de la fonction main() elle-même. Obtenir l'adresse d'une fonction est similaire à ce qu'on fait pour une variable : `printf("%p\n",&main);` Où est placé le code de cette fonction ?

▷ **Question 4.** Ajoutez une variable de type `char*` dont le contenu est l'adresse d'un bloc alloué dynamiquement avec l'appel `malloc(512);`. Où est placée cette variable ? Où est placé le bloc alloué ?