TP7 : Allocation mémoire et mise au point de programmes C

CSH : Initiation au C et au shell

Première année

▶ Partie 1: Mise au point de programmes C

Cette première partie a pour objectif de vous donner quelques 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 consistent 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.

★ Exercice 1: 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.

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 2: le debugger GNU : gdb (utilisation de base)

Nous utiliserons comme premier exemple le programme `boucle.c` ci-contre (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).

▷ **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.

boucle.c

```
#include <stdio.h>                                          1
#include <stdlib.h>                                         2
                                                            3
int *tab = NULL;                                            4
                                                            5
void initialise(int n) {                                    6
  char i = 0;                                               7
                                                            8
  for (i = 0; i <= n; i++);                                 9
    {                                                       10
       tab[i] = 1;                                          11
    }                                                       12
}                                                           13
                                                            14
int main() {                                                15
  printf("Debut\n");                                        16
  tab = malloc(10000*sizeof(int));                          17
  initialise(10000);                                        18
  printf("Fin\n");                                          19
  printf("tab[0]=%d;tab[100]=%d\n",                         20
         tab[0],tab[100]);                                  21
                                                            22
  return 0;                                                 23
}                                                           24
```

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 3: 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.

fact.c

```c
#include <stdio.h>

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;
}
```

★ Exercice 4: 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 :

```
==394== LEAK SUMMARY:
==394==    definitely lost: 0 bytes in 0 blocks.
==394==      possibly lost: 0 bytes in 0 blocks.
==394==    still reachable: 400 bytes in 1 blocks.
==394==         suppressed: 0 bytes in 0 blocks.
==394== Reachable blocks (those to which a pointer was found) are not shown.
==394== To see them, rerun with: --show-reachable=yes
```

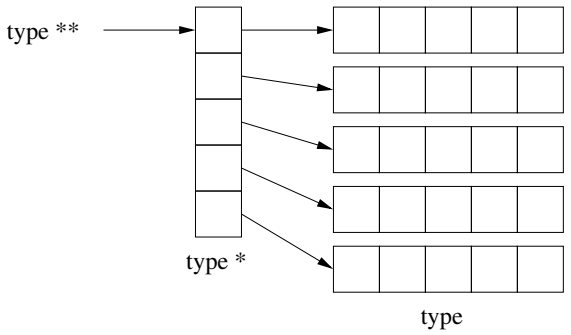
Il y a donc un bloc de mémoire (de 400 octets) obtenu par `malloc`, mais jamais restitué au système avec `free`. Ajoutez l'option nécessaire pour voir lequel et corrigez le problème.

▶ Partie 2: Allocation de tableaux multidimensionnels

Nous allons maintenant appliquer ces méthodes de debug pour mettre au point des programmes manipulant la mémoire.

En C, l'allocation dynamique d'un tableau multidimensionnel se fait en deux étapes :
- allocation d'un tableau de pointeurs dont la taille correspond au nombre de lignes souhaité
- allocation de chacune des lignes en utilisant le tableau de pointeurs précédent pour stocker leur adresse

Ensuite, l'accès au tableau se fait comme d'habitude en C, en utilisant l'opérateur [] pour préciser l'indice dans chacune des dimensions. La seule difference avec un tableau alloué de manière statique est le type des objets manipulés : dans le cas dynamique ce sera un tableau de pointeurs, dans le cas statique ce sera un tableau de tableaux (les tableaux sont réellement stockés les uns après les autres en mémoire).



Structure d'un tableau bidimensionnel dynamique

Exemple : allocation d'un tableau de 10*10 doubles (sans aucune vérification)

```
double **tableau;

tableau = (double **) malloc(sizeof(double *)*10);
for (i=0; i<10; i++)
    tableau[i] = (double *) malloc(sizeof(double)*10);
```

Idem avec libération en cas d'erreur

```
double **tableau;

tableau = (double **) malloc(sizeof(double *)*10);
if (tableau != NULL) {
  i=0;
  erreur=0;
  while ((i<10) && !erreur) {
    tableau[i] = (double *) malloc(sizeof(double)*10);
    if (tableau == NULL)
      erreur = 1;
    else
      i++;
  }
  if (erreur) {
    while (i) {
      i--;
      free(tableau[i]);
    }
  }
}
```

Initialisation des éléments

```
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        tableau[i][j] = 0;
```

Vous trouverez dans le répertoire ***partie_2*** plusieurs programmes à compléter. Vous pouvez compiler et tester tous les exercices de la séance avec la commande `make test`

Chaque test correspond à un programme (le nom du programme s'affiche avant ECHEC ou SUCCES) que vous pouvez exécuter independemment pour déterminer vos erreurs. Le test <toto> est passé si votre programme `toto` affiche exactement la même chose que ce qui se trouve dans le fichier `toto.result`. N'hésitez pas à consulter le contenu des fichiers fournis pour comprendre le fonctionnement de l'ensemble.

★ Exercice 1: Vecteurs

Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type vecteur. Complétez ***vecteur.c*** qui contient toutes les fonctions de gestion de vecteur que nous voulons implémenter. Le fichier ***vecteur.h*** contient toutes les informations nécéssaires : description des fonctions et déclaration du type vecteur. Une fois les fonctions complétées, tapez `make vecteur_testbase` pour compiler, `./vecteur_testbase` pour executer le programme de test et make test pour tester si le résultat du programme de test est correct (s'il affiche la même chose que ce qui se trouve dans ***vecteur_testbase.result***).

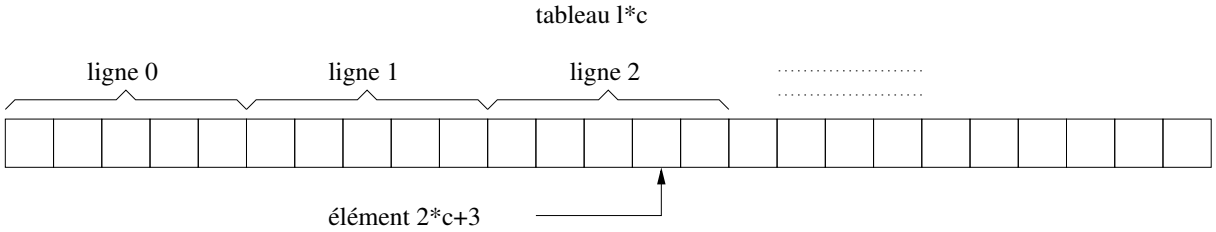
★ Exercice 2: Matrices

Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type matrice, où les matrices sont représentées par un tableau bidimensionel dynamique. Complétez le fichier ***matrice.c*** qui contient toutes les fonctions de gestion de matrice que nous voulons implémenter. Le fichier ***matrice.h*** contient toutes les informations nécessaires : description des fonctions et déclaration du type matrice. Une fois

les fonctions complétées, tapez `make matrice_testbase` pour compiler, `./matrice_testbase` pour exécuter le programme de test et `make test` pour tester si le résultat du programme de test est correct.

★ **Exercice 3: Matrices linéaires**

Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type matrice, où les matrices sont représentées par un seul tableau unidimensionel. Dans ce cas, les lignes de la matrice sont stockées les unes après les autres et la fonction d'accès va devoir faire la traduction d'un couple d'indices vers un unique indice dans le tableau (l'avantage est que l'on économise une indirection ainsi que le coût des structures de gestion mémoire associées à chaque `malloc` par le système). La figure suivante décrit l'organisation de nos matrices en mémoire :



Complétez le fichier ***matrice_lineaire.c*** qui contient toutes les fonctions de gestion de matrice que nous voulons implémenter. Le fichier ***matrice_lineaire.h*** contient toutes les informations nécessaires : description des fonctions et déclaration du type matrice. Une fois les fonctions complétées, tapez `make matrice_lineaire_testbase` pour compiler, `./matrice_lineaire_testbase` pour exécuter le programme de test et `make test` pour tester si le résultat du programme de test est correct.

★ **Exercice 4: Vérifier les bornes**

Reprenez les trois exercices précédents en ajoutant un test dans la fonction d'accès permettant de renvoyer NULL si les indices demandés sont en dehors des bornes de l'objet concerné. Les fichiers à completer sont ***vecteur_verif.c***, ***matrice_verif.c*** et ***matrice_lineaire_verif.c***. Les commandes sont analogues aux exercices précédents.

★ **Exercice 5: Réallocation dynamique**

Reprenez les trois premiers exercices en ajoutant une réallocation dynamique des objets dans la fonction d'accès. Le comportement de cette fonction doit alors être le suivant :

- si un des indices d'accès est négatif retourner `NULL` ;
- si les indices sont dans les bornes de l'objet retourner le pointeur d'accès au bon élément ;
- si un des indices dépasse les bornes de l'objet tenter de réallouer plus de mémoire et renvoyer le pointeur d'accès au bon élément si la réallocation réussit et `NULL` sinon.

Pour la réallocation, on pourra utiliser au choix un `malloc` d'un bloc plus gros suivi d'une copie ou bien l'appel système `realloc` dont on obtient la description avec la commande `man realloc`.

Complétez ***vecteur_dynamique.c***, ***matrice_dynamique.c*** et ***matrice_li neaire_dynamique.c***.

★ **Exercice 6: Opérations mémoire.** Implémentez les fonctions de manipulation mémoire suivantes :

- `my_memcpy` : copie d'une zone en mémoire de la même manière que `memcpy` (cf. man) ;
- `my_memmove` : copie d'une zone en mémoire avec recouvrement possible. (cf. `man memmove`) ;
- `is_little_endian` : renvoie vrai si l'architecture cible utilise la convention little endian pour la représentation des entiers en mémoire ;
- `reverse_endianess` : renvoie la valeur passée en argument avec ses octets inversé.

Le fichier à completer est ***memory_operations.c***.