TD-TP 1: Rappels sur l'allocation dynamique de mémoire. Introduction à quelques outils (indispensables) de développement.

M1 Informatique, Université Grenoble Alpes

2019

L'archive (`tp1.tar.gz`) contenant les fichiers utilisés pour cette séance se trouve sur Moodle. [1].

I. Les travaux pratiques (développement) en CSE-PC

Vous démarrez une matière notoirement *technique*. Elle compte pour 6ECTS et son succès repose énormément sur le travail pratique. Vous aurez à coder ce semestre!

Toutefois, il ne s'agit pas de coder *n'importe comment*. Il faudrait donc non seulement s'approprier les concepts (les problèmes) et comprendre ce qu'il faut faire, mais également le faire *de la bonne manière*.

Question I.1: Le processus de développement *D'après vous, quelles sont les phases lors du développement d'un logiciel (d'un TP en CSE-PC) ? Est-il intéressant d'automatiser certaines phases? Connaissez-vous des outils qui peuvent vous aider?*

II. Allocation dynamique de mémoire et rappels sur les pointeurs

Considérons l'exemple simple donné ci-dessous :

Listing 1 Exemple minimal

```
1  #include <stdio.h>
2  int main(){
3      int a, b=10;
4      a=5;
5      printf("Somme de a+b: %d\n", a+b);
6      exit(0);
7  }
```

Question II.1: *Où se trouvent les variables `a` et `b`? Combien d'octets occupent-elles?*

[1] Pour extraire les fichiers, utiliser la commande `tar zxvf tp1.tar.gz`. Pour plus de détails sur les archives `tar`, faire `man tar`.

Question II.2: *Quelle est la valeur de* `a` *à la ligne 3? Et à la ligne 4?*

Question II.3: *Qui s'occupe de la gestion des emplacements des variables* `a` *et* `b`*? Le programmeur? Le système?*

Question II.4: *Peut-on connaître l'emplacement* exact *de la variable* `a`*?*

Considérons maintenant l'exemple suivant :

Listing 2 Exemple avec pointeurs

```
1  #include <stdio.h>
2  int main(){
3      int a, b=10;
4      a=5;
5
6      int* c;
7      c = &a;
8      printf("L'adresse de a: %p\n", c);
9
10     int valeur_mystere = *c;
11     printf("Valeur mystere: %d\n", valeur_mystere);
12
13     exit(0);
14 }
```

Question II.5: *A la ligne 6, une nouvelle variable est déclarée. Quel est son type? Combien d'octets occupe-t-elle? Quelles valeurs est elle destinée à recevoir?*

Question II.6: *Que se passe-t-il à la ligne 7?*

Question II.7: *Que se passe-t-il à la 10?*

Passons à l'exemple 3 :

Listing 3 Un peu d'arithmetique de pointeurs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct {
5    int op1;
6    int op2;
7    float param;
8    int bool;
9  } mystruct_t;
10
11 int main(){
12     int a=5, b=10;
13     int arr_int[2];
14     mystruct_t arr_mystruct[5];
```

```
15
16      int* c = arr_int;
17      *c = a;
18      *(c+1) = b;
19
20      mystruct_t ms = {1, 2, 0.10, 0};
21      mystruct_t* s = arr_mystruct;
22      for (int i = 0; i < 5; i++) {
23         *(s+i) = ms;
24         ms.op1+=1;
25      }
26
27      printf("Element 3 = {%d, %d, %f, %d}\n", (s+2)->op1, (s+2)->op2, (s+2)->param, (s+2)->
            bool);
28      exit(0);
29  }
```

Question II.8: *Quel est le type de* `arr_int`*? Et de* `arr_mystruct`*?*

Question II.9: *Que fait le programme lignes 16-18?*

Question II.10: *Que fait le programme lignes 21-25?*

Question II.11: *Expliquer le fonctionnement des opérateurs* `'+'` *et* `'-'` *avec les pointeurs.*

Question II.12: *Que se passerait-il avec*

```
*(int*)((char*)c+2)=2;
```

Question II.13: *Et avec*

```
*(mystruct_t*)c=ms;
```

Considérons le dernier exemple de cette section :

Listing 4 Un peu d'arithmetique de pointeurs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int *a, *b;
6      a = malloc(sizeof(int));
7      b = malloc(sizeof(int));
8      *a = 5;
9      *b = 10;
10
11     printf("Somme des deux valeurs: %d\n", *a+*b);
12     exit(0);
13 }
```

Question II.14: *Commentez les différences par rapport au tout premier exemple. A quoi sert la fonction* `malloc`*?*

Question II.15: *Pour un développeur rigoureux, ne manque-t-il pas qqchose à la fin du programme?*

III. La pile et le tas

La *pile* et le *tas* sont deux espaces mémoire disticts, gérés par le système, au sein de chaque processus afin qu'il puisse stocker ses données (variables)[2].

- Dans la pile sont allouées, de manière *automatique*, les variables locales des fonctions. Comme leur taille est connue lors de la compilation[3], la mémoire est automatiquement réservée quand la fonction est appelée, et libérée, quand la fonction termine. En d'autres termes, la pile est un endroit do stockage temporaire de variables ayant comme portée la durée d'exécution de la fonction.
- Le tas est utilisé pour stocker des données dont la taille n'est pas connue lors de la compilation. Typiquement, la taille d'une matrice qui va être utilisée au ein d'un programme peut dépendre des arguments passés au lancement. Cette mémoire doit être gérée de manière explicite par le programmeur, à l'aide de fonctions standard comme `malloc` et `free` (voir `man malloc`).

Considerer le code suivant (fourni dans `ex1.c`) et répondre à la question:

```
#include <stdio.h>

int min(int a, int b, int c){
  int tmp_min;
  tmp_min = a <= b ? a : b;
  tmp_min = tmp_min <= c ? tmp_min : c;
  return tmp_min;
}

int main(){
  int min_val = min(3, 7, 5);
  printf("The min is: %d\n", min_val);
  exit(0);
}
```

Question III.1: *Dans quel segment mémoire (pile ou tas) sont allouées les variables* `a`*,* `b` *et* `c`*? Quand la mémoire qui leur est allouée est libérée? Et la variable* `tmp_min` *?*

[2] L'organisation exacte de la mémoire d'un processus va être vue en cours plus tard dans le semestre.
[3] Le processus de génération d'un exécutable va également êtr traité plus tard pendant le semestre.

Considérons maintenant (`ex2.c`):

```
int* vect_sum(int *v1, int *v2, int size){
  int *r, i;
  r = malloc(sizeof(int) * size);
  for(i = 0; i < size; i++){
    r[i] = v1[i] + v2[i];
  }
  return r;
}

int main(){
  int v1[] = {1, 2, 4, 7};
  int v2[] = {3, 4, 9, 2};

  int *p_result = vect_sum(v1, v2, 4);
  /*imprime le contenu du vecteur */
  print_vect(p_result, size);
  exit(0);
}
```

Question III.2: *Quelle valeur contient* `r` *après l'appel à* `malloc()` *dans la fonction* `vect_sum()`*? Dans quel segment mémoire est stockée cette valeur?*

Question III.3: *Que fait l'instruction d'affectation suivante:* `r[i] = v1[i] + v2[i];` *? Cette affectation provoque une écriture en mémoire. Dans quel segment mémoire?*

Question III.4: *Ecrire un programme qui a le même comportement mais sans utiliser* `malloc()`*. Il est possible que vous ayez besoin de changer les paramètres de la fonction* `vect_sum()`*.*

Question III.5: *Quel est le cycle de vie d'une variable allouée sur la pile? Et d'une variable allouée sur le tas?*

IV. Accès mémoire illégaux

Avec une allocation mémoire correcte, chaque variable a sa propre place en mémoire. En utilisant les pointeurs, il est toutefois possible d'accéder à des emplacements qui ne correspondent pas à des varibles allouées. Dit autrement, il est possible de vouloir accéder une adresse mémoire sans que cette adresse soit allouée (corresponde) à une variable. Si un programme essaie d'effectuer un tel accès, il peut être tué *par le système* avec le signal `SIGSEGV`. L'accès correspondant est appelé *accès mémoire illégal*.

Question IV.1: *Dans le code qui suit, quelles correspondent à des accès mémoire illegaux? Quels accès provoqueraient des avertissements de la part d'un compilateur "pénible"?*

```
1. int *pa = 2;
2. *pa = 34;
3. int b = 4, *pb = &b;
4. *pb = 5;
5. int *pc;
6. printf("pc is equal to %d\n", pc);
7. printf("*pc is equal to %d\n", *pc);
8. pc = malloc(sizeof(int));
9. *pc = -2;
10. pa = pc;
11. free(pa);
12. pc = -4;
```

V. Les Makefiles

Dans l'archive qui vous est fournie, il y a un *Makefile*. Les Makefiles sont utilisés pour automatiser le processus de compilation à l'aide de l'outil `make`. Surtout, les Makefiles permettent de compiler un ensemble de fichiers afin de générer un exécutable, en gérant les dépendances et en ne compilant que ce qui est nécessaire.

Si vous ne connaissez pas les Makefiles, celui qui vous est fourni contient des commentaires pour vous aider à comprendre de quoi il s'agit. N'hésitez pas à vous former sur Internet.

Pour vérifier votre compréhension du fonctionnement des Makefiles, ouvrez celui qui vous est fourni et répondez aux questions suivantes.

Question V.1: *Lister les commandes qui vont être exécutées en tapant*

```
make ex1.run
```

Question V.2: *Lister les commandes qui vont être exécutées en tapant*

```
make ex2.run
```

Expliquer à quoi correspondent les variables '`$@`' et '`$<`'.

Question V.3: *Lister les commandes qui vont être exécutées en tapant*

```
make prog_0.run
```

A quoi correspond '`%`' dans une règle?

Question V.4: *Lister les commandes qui vont être exécutées en tapant*

```
make all
```

VI. Gdb

`gdb` est un outil de débogage. Il permet l'exécution pas-à-pas d'un programme et l'exploration de la mémoire du processus correspondant (pile, tas, registres, ...).

Question VI.1: *Nous vous fournissons un tutoriel gdb simple que nous vous conseillons de suivre. Ouvrez `gdb-tutorial.c` et suivez les instructions.* [4]

VII. Valgrind

`valgrind` est un utilitaire qui détecte des erreurs qui surviennent à l'exécution. Il simule l'exécution d'un exécutable et détecte des erreurs comme les accès mémoire illégaux.

Pour l'utiliser:

```
$ valgrind ./my_executable
```

Nous vous fournissons quelques programmes (prefixe "`prog_`") qui sont syntaxiquement corrects mais génèrent des erreurs à l'exécution.

Question VII.1: *Utiliser `valgrind` pour trouver les erreurs dans les programmes C fournis.*[5] `valgrind` *est de manière générale très bavard: recopier la sortie qui concerne les erreurs et expliquer. (Si le programmes sont compilés avec l'option `-g`, valgrind dit dans quel fichier et à quelle ligne se produit l'erreur.). Considérer également les options suivantes :*

- *Vous pouvez utiliser `gdb` et `valgrind` de manière conjointe. Voir http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver-simple*
- *--leak-check=yes (avoir de l'information sur de la mémoire non libérée et perdue);*
- *--show-reachable=yes (information sur de la mémoire non libérée mais utilisable).*

VIII. AddressSanitizer (ASan)

`AddressSanitizer` est un outil qui peremt de détecter les accès mémoire illégaux. Son fonctionnement diffère de celui de `valgrind`. `AddressSanitizer` instrumente le code source de l'application et de ce fait requière sa recompilation.

`AddressSanitizer` est intégré dans `gcc` depuis la version 4.8. Il est activement utilisé dans le développement des navigateurs `chromium` et `firefox`.

Pour utiliser `AddressSanitizer`, compiler le code avec les flags suivants :

```
$ gcc -g -fsanitize=address my_file.c -o my_exec_file
```

[4] `gdb` sera un outil indispensable pour le TP noté mémoire qui démarre la semaine prochaine.
`gdb` fournit un mode textuel de base. Nénamoins il peut être utilisé à travers d'interfaces graphiques comme `ddd`. Pour plus de détails voir `https://sourceware.org/gdb/wiki/GDB Front Ends`

[5] Garder une copie des programmes initiaux pour les utiliser dans l'exercice suivant.

EXécuter le code comme d'habitude.

Question VIII.1: *Observer les erreurs des programmes "prog_.c" relevées par* `AddressSanitizer`*. Utiliser les versions initiales i.e. erronnées :)*

Comparer Valgrind et AddressSanitizer: Les deux outils sont utiles. De point de vue du fonctionnement (couverture des erreurs), les deux peuvent repérer des erreurs que l'autre outil ne détecte pas (cela arrive dans des cas spécifiques que vous n'aurez a priori pas à gérer). Par rapport à la performance, toutefois, `AddressSanitizer` est bien plus rapide que `Valgrind`. Le coût à payer est la recompilation de l'application.

Question VIII.2: *Pour approfondir vos connaissances sur* `AddressSanitizer`*, vous pouvez aller voir:*

- *The Github repository:* `https://github.com/google/sanitizers/wiki/AddressSanitizer`
- *Publication principale :* `https://research.google.com/pubs/pub37752.html`

– Bonus –

Pour les curieux/motivés.

IX. Fonctions récursives

```
int power(int a, int n){
  if( n != 0 )
    return a*power(a, n - 1);
  else
    return 1;
}
```

Question IX.1: *Estimer en gros combien de mémoire serait nécessaire pour calculer* `power(2, 3)`.

Question IX.2: *Nous vous fournissons* `rec.c`*. Essayer de valider votre estimation.*

Pour ce faire, utiliser les fonctions intégrées[6] *gcc function* `void * __builtin_frame_address (unsigned int level)` *retourne l'adresse de retour pour le cadre d'une fonction (avec* $level = 0$*, retourne l'adresse de retour de la fonction courante)*[7]

[6] Les fonctions intégrées ne sont pas incluses dans le standard C

[7] See `https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html`

X. Plus sur Gdb

Question X.1: *Considérons de nouveau les programmes de l'exercice I. Utiliser* `gdb` *afin de visualiser l'état ainsiq que l'évolution de la pile et du tas du programme en exécution.*

Voici quelques commandes gdb utiles :

- *x/nfu addr*
  x affiche la mémoire à partir de l'adresse addr; n, f, et u sont des paraèmtres optionnels pour formatter l'affichage.
  - ***n**: Entier décimal (1 par défaut) pour dire combien d'unités mémoire vont être affichées, les unités étant définies par u*
  - ***f**: Format de l'affichage: 's' (chaine de caractères se terminant par 0), ou 'i' (instruction machine).La valeur par défaut est 'x' pour hexadecimal.*
  - ***u**: Taille d'unité : 'b' (Bytes), 'h' (Halfwords – 2 bytes), 'w' (Words – four bytes – default), 'g' (Giant words – eight bytes).*
- *info frame*
  information sur les cadres (contexte par fonction) de la pile.
- *p $sp*
  donne la valeur du registre de pile
- *x/5i $pc-6*
  imprime 5 instructions 6 mots avant le pointeur de programme courant

Question X.2: *Essayer gdb sur le programme recursif fourni.*