Correction de l'examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 9 janvier 2019

Exercice 1 : Fonctions simples (6 points)

1.
```
/*@ requires *pa et *pb valides
    assigns *pa et *pb
    ensures permute les valeurs de *pa et *pb */
void swap(int *pa, int *pb) {
  int tmp;
  tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

2.
```
/*@ requires *pc valide
    assigns *pc
    ensures permute les valeurs des champs de *pc */
void swap_couple(struct couple *pc) {
  double tmp;
  tmp = pc->first;  /* ou bien (*pc).first */
  pc->first =
  *pa = *pb;
  *pb = tmp;
}
```

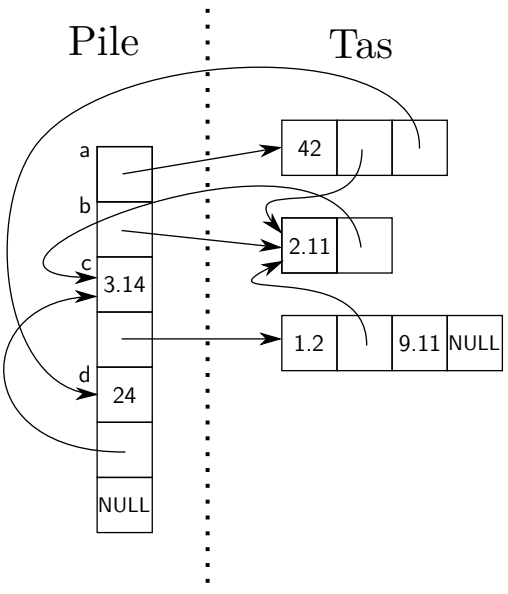
3.
```
/*@ requires t tableau de taille au moins size
    assigns rien
    ensures retourne une copie de t de taille size */
int *array_copy(int t[], int size) {
  int *res;
  int i;
  res = (int*) malloc(size * sizeof(int));
  for (i = 0; i < size; i = i + 1)
    res[i] = t[i];
  return res;
}
```

Remarque : il est nécessaire d'allouer de la mémoire dans le tas avec `malloc` sinon le tableau est perdu après la sortie de l'appel à `array_copy`. (Pas de `int res[size];` donc.)

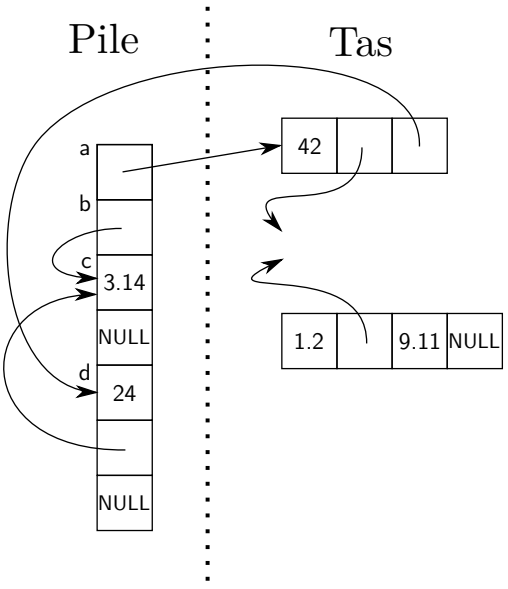
4.
```
/*@ requires s chaine de caracteres bien formee
                              (se termine par '\0')
    assigns rien
    ensures retourne une copie de s */
char *string_copy(char *s) {
  char *res;
  int i;
  int size;
  size = 0;
  /* on calcule la taille de la chaine
     en augmentant size tant qu'on a pas atteint '\0' */
  while (s[size] != '\0')
    size = size + 1;
  size = size + 1; /* pour stocker le '\0' final */
  res = (char*) malloc(size);
      /* on peut utiliser le fait que sizeof(char) == 1 */
  for (i = 0; i < size; i = i + 1)
    res[i] = s[i];
  return res;
}
```

Exercice 2 : Représentation mémoire (5 points)

1.



Remarque : tous les points étaient attribués même si la pile n'était pas distinguée du tas.

2. 

3. Des pointeurs ont pour valeur une adresse qui vient d'être libérée (`a->path` par exemple). Cette zone mémoire est susceptible d'être réutilisée pour d'autres données.

   Il y a une fuite mémoire : le bloc de deux **`struct bucket`** alloués n'est plus accessible mais n'a pas été libéré.

Exercice 3 : Modularité (4 points)

1. `Server` dépend de l'interface de `Common`. `Client` dépend des interfaces de `Common` et `Graphics`. Tous les modules dépendent de leur propre interface.
2. `gcc -Wall -Wextra -ansi -c Server.c`
3. `gcc -ansi Server.o Common.o -o server`
4. Uniquement le module `Graphics`.
5. Il faut recompiler `Common`, `Client` et `Server`.
6.
```
CC=gcc -Wall -Wextra -ansi

common.o: common.h
server.o: server.h common.h
graphics.o: graphics.h
client.o: client graphics.h common.h

server: server.o common.o
	$(CC) $^ -o $@
```

```
client: client.o common.o graphics.o
_________$(CC) $^ -o $@
```

Exercice 4 : Étude de fonctions (5 points)

1. `x` et `n` sont des **`int`**, donc l'opération `-x * x / n` se fait sur les **`int`**. Pour `x` $= 1$ et `n` $= 2$, elle vaut donc 0. Par conséquent, on retourne 2^0, c'est-à-dire 1.

2. Les **`unsigned char`** sont des entiers non-signés sur un octet. Ils prennent donc des valeurs entières entre 0 et 255.

   La première case du tableau, qui vaut 1 au départ, vaut donc la partie entière de 1.5 ensuite, c'est-à-dire 1.

   La dernière case du tableau, qui vaut 200 au départ, est d'abord multipliée par 1.5, ce qui donne 300. On stocke à la place ce 300 mais modulo 256 puisqu'on a un **`unsigned char`**. On stocke donc 44. ($300 = 44 \mod 256$.)

   Au final, le tableau contient donc les valeurs `1`, `3`, `15`, `150`, `44`.

3. `b = 1` est une affectation, sa valeur est celle de la valeur affectée, donc ici 1. Par conséquent la condition du test est toujours vrai et la fonction renvoie toujours `x + 1`. On renvoie donc `11`.

   Remarque : comme cela dépasse le cadre vu en cours, des points étaient attribués si on écrivait que le code contient une erreur et ne compile pas.

4. Si `b` vaut 1, sa négation `!b` vaut 0 et le test est donc faux. Néanmoins, seule la première instructions suivant le **`if`** lui est rattachée ; l'instruction `*py = *py + 1;` est donc exécutée dans tous les cas. L'appel va donc laisser la valeur de `x` inchangée à 42 et va incrémenter la valeur de `y` à 25.

5. Dans la ligne **`char **res = (char**) malloc(n * sizeof(char));`** on attribue `n` cases mémoires de taille **`char`** pour y stocker `n` pointeurs de taille **`char*`**. Comme a priori la taille de **`char*`** est plus grande que **`char`**, il n'y aura pas assez de mémoire allouée, et lors de la boucle **`for`**, on va écrire en dehors de la zone allouée.

   Le comportement est alors indéfini : on peut par exemple obtenir une erreur de segmentation immédiatement, ou bien le programme peut continuer sans rien dire. Dans ce dernier cas, la zone mémoire dans laquelle on a mis la fin du tableau peut être allouée à d'autres données, ce qui entraînera vraisemblablement des bugs très difficiles à diagnostiquer. Alternativement, cela peut également entraîner le fait d'avoir des démons qui sortent du nez (cf. `https://en.wikichip.org/wiki/nasal_demons`).