TD3: Mémoire statique – Correction

Module Sys1

Objectifs pédagogiques:

- Comprendre la durée de vie des variables (E1, E2, E6);
- Comprendre la visibilité des variables et fonctions (E1);
- Comprendre la différence entre un tableau et un pointeur (E3);
- Manipulation de pointeurs (E2, E3, E4, E5, E7).

Les exercices et questions indiquées comme optionnelles ne sont là que pour votre culture et pour amuser les élèves qui le souhaitent. Ne les faites pas si ce n'est pas un jeu pour vous. Ils ne seront pas corrigés en classe, et ne donneront lieu à aucune évaluation.
Des questions issues d'exercices non-optionnels peuvent être posées en examen.
À préparer avant la séance : Exercice 1, Exercice 4 et Exercice 5 question 1.

Réponse

Ce TD se fait au tableau, sans machine. C'est l'occasion de revenir sur les notions de pile d'exécution et de cadres de pile (qui contiennent les paramètres et autres variables locales). C'est un rappel de cours, mais c'est très souvent nécessaire.

L'exercice 2 permet d'explorer des erreurs classiques que l'on fait souvent quand on commence à manipuler des pointeurs. Il est important d'expliquer pourquoi ça ne fonctionne pas, en faisant des schémas mémoires de l'état de la pile pour bien comprendre.

Certaines questions nécessitent des éléments que nous n'avons pas encore vu en cours. Il peut être profitable de relire ces exercices dans deux ou trois semaines.

Fin réponse

★ Exercice 1: Visibilité et durée de vie (à préparer avant).

Supposons les deux fichiers suivants :

fichier1.c

```c
#include <stdio.h>

void func3(void);
int pi = 3;
static char c = 'c';

static void func1(int a) {
    printf("Life be%came interesting in %d\n", c, a);
}
void func2(void) {
    static int year = 1972;
    int g = 0;

    func1(year - g);
    func3();
}
int main(void) {
    func2();
    return 0;
}
```

fichier2.c

```c
#include <stdio.h>

extern int pi;
void func3(void) {
    int i;
    for (i = 0; i < pi; i++) {
        printf("%d", pi);
    }
    printf("\n");
}
```

Définissez chaque fonction et variable en terme de visibilité (globale, fichier, limité à une fonction ou à un bloc) et durée de vie dans le tableau suivant :

	Visibilité	Durée de vie
`printf`	Globale	Fin du programme
`main`	Globale	Fin du programme
`func1`	Fichier	Fin du programme
`func2`	Globale	Fin du programme
`func3`	Globale	Fin du programme
`pi`	Globale	Fin du programme
`c`	Fichier	Fin du programme
`year`	Fonction `func2`	Fin du programme
`a`	Fonction `func1`	Fin d'appel à `func1`
`g`	Fonction `func2`	Fin d'appel à `func2`
`i`	Fonction `func3`	Fin d'appel à `func3`

Réponse

Cet exercice est une excuse à rappel de cours sur le schéma mémoire d'un processus, et sur les notions de visibilité et durée de vie. Il faut au moins prendre le temps de refaire le schéma suivant pour comprendre où sont stockées les différents types de variable.

Fin réponse

★ **Exercice 2: Erreurs classiques.**

▷ **Question 1:** Quel est le problème 1 ci-dessous? Comment le corriger, sans changer la ligne 8?

▷ **Question 2:** Quel est le problème avec le programme 2 ci-dessus ? Comment le corriger ?

▷ **Question 3:** Quel est le problème avec le programme 3A à gauche? Pourquoi cela fonctionne avec 3B?

▷ **Question 4:** Expliquez le problème du programme 4 (demandez-vous où est stockée la chaîne `"Good"`).

Programme 1

```
1  #include <stdio.h>
2
3  static int* func(int n) {
4      n = n + 10;
5      return &n;
6  }
7  int main(void) {
8      int* a;
9
10     a = func(19);
11     printf("%d\n", *a);
12     return 0;
13 }
```

Programme 2

```
1  #include <stdio.h>
2
3  int main(void) {
4      char* nom;
5
6      printf("Quel est votre nom ?");
7      scanf("%s", nom);
8
9      return 0;
10 }
```

Réponse

Le **Programme 1** retourne l'adresse d'une variable locale, qui se trouve donc dans la pile: il s'agit du paramètre `n` (oui, les paramètres sont des variables locales presque comme les autres).
On peut corriger ça en passant l'adresse de la variable `a` à `func()` pour qu'elle puisse la modifier directement. Cette adresse sera valide, car elle est dans le cadre de pile de `main()`, qui restera valide jusqu'à la fin de la fonction `func()`.
Une autre façon de corriger (respectant l'énoncé, cette fois, càd sans changer la ligne 8), il faut une solution pour rendre permanent la chose où on stocke n+10. On peut créer une nouvelle variable locale statique pour ça.
Moralité: utiliser l'adresse des variables locales n'est pas interdit, tant qu'on est sûr qu'on ne va pas s'en servir après la destruction du cadre de pile les contenant. Faire des dessins aide à s'en assurer.

```
1  #include <stdio.h>
2
3  static void func(int* n) {
4    *n = *n + 10;
5
6  }
7  int main(void) {
8    int a = 19;
9
10   func(&a);
11   printf("%d\n", a);
12   return 0;
13 }
```

```
1  #include <stdio.h>
2
3  int main(void) {
4      char nom[512];
5
6      printf("Nom : ");
7      scanf("%s", nom);
8
9      return 0;
10 }
```

Le problème du **Programme 2** est différent. Cette fois, `nom` est un "pointeur pendouillant" (*dangling pointer* en anglais), c'est-à-dire un pointeur vers une zone invalide. En effet, on ne lui a jamais donné de valeur explicitement, donc il vaut n'importe quoi.
Ce qui fait que quand `scanf` va recopier les lettres tapées au clavier une à une, elle va écrire n'importe où en mémoire. Si on a de la chance, on va tenter d'écrire à un endroit interdit et l'OS va tuer le processus. Si on n'a pas de chance, on va corrompre une donnée du programme et créer un bug très difficile à diagnostiquer correctement.
La solution la plus simple consiste à réserver la place explicitement, en changeant `nom` en un tableau (ligne 4 ci-contre). Ainsi, `scanf` va écrire dans les cases réservées, comme attendu. Cette solution n'est pas parfaite, car si l'humain a un nom de plus de 512 caractères (ou l'esprit taquin), on écrit au dela du tableau réservé.

Fin réponse

Programme 3A

```
1  #include <stdio.h>
2
3  int* func(int n) {
4      int array[2] = { 14, 15 };
5      if (n == 0) {
6          return array;
7      } else {
8          return NULL;
9      }
10 }
11 int main(int argc, char* argv[]) {
12   int *t = func(0);
13   printf("t[0]=%d; t[1]=%d\n", t[0], t[1]);
14 }
```

Programme 3B

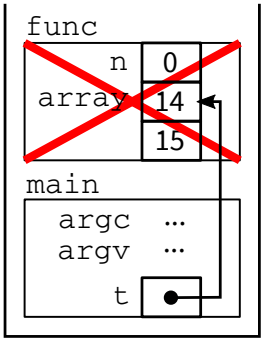
```
1  #include <stdio.h>
2
3  int* func(int n) {
4      static int array[2] = { 14, 15 };
5      if (n == 0) {
6          return array;
7      } else {
8          return NULL;
9      }
10 }
11 int main(int argc, char* argv[]) {
12   int *t = func(0);
13   printf("t[0]=%d; t[1]=%d\n", t[0], t[1]);
14 }
```

Réponse

La différence entre les deux codes donne un indice fort. A droite, la variable array est marquée `static`. Donc, elle n'est pas dans le cadre de pile de `func()` (comme c'est le cas à gauche), mais dans le segment DATA du processus. Et c'est pour ça que ça marche: le segment DATA est stable dans le temps, tandis que le cadre de pile va être détruit au moment du `return`, ligne 6.

Le dessin ci-contre est important pour comprendre. Il montre ce qui se passe dans le cas de gauche juste après le return de la ligne 6, qui rend la main à la ligne 12. Le tableau `t` pointe bien sur `array`, mais c'est dans une zone de mémoire recyclée. Le prochain appel de fonction va l'écraser.

Moralité: quand on utilise l'adresse d'une variable en C, il faut s'interroger sur la durée de vie de cette variable. Va-t-elle rester en place jusqu'à ce que j'ai fini de m'en servir ?



Fin réponse

Programme 4

```
1  void edit(char* s) {
2    s[0] = 'B'; // segfault
3    s[1] = 'a';
4    s[2] = 'd';
5    s[3] = '!';
6  }
7  int main(void) {
8    edit("Good");
9
10   return 0;
11 }
```

Réponse

Aucun warning à la compilation, même avec `-Wall -Wextra`, mais "Erreur de segmentation" à l'exécution. C'est cruel. Les outils de debug que l'on verra dans deux semaines sont pas tous utiles. `gdb` nous dit juste où précisément se trouve le segfault, mais rien sur la cause.

`valgrind` est utile, lui. Il nous dit "Bad permissions for mapped region at address 0x10A004". En effet, la chaîne dont l'adresse est passée en paramètre à `edit`, est stockée dans une section de données interdite en écriture (`.rodata`), ainsi quand on essaye d'écrire un `R` à la place du `C` le programme est stoppé. Bon, valgrind parle à celleux qui savent, mais déjà il dit ce qu'il en est.

Il est possible d'utiliser le warning `-Wwrite-strings` (voir man gcc) pour détecter ce type de problème. Avec ça, on voit le message suivant: "error: passing argument 1 of 'edit' discards 'const' qualifier from pointer target type [-Werror=discarded-qualifiers]" qui est parfaitement correct et descriptif du problème, même si les outils C sont toujours aussi peu pédagogiques. Ici, gcc nous dit que la chaîne good est constante (dans le sens immutable) et qu'il est potentiellement dangereux de la passer à une fonction dont le paramètre n'est pas `const char*`. Mais si on change le prototype de la fonction, alors gcc ne nous laisse plus faire la ligne 2, qui change une chaîne constante. La morale est qu'on peut détecter ces problèmes à la compilation en C, en passant des kilomètres de flags `-W??` à la compilation, et en s'astreignant à respecter les conseils que le compilateur donne alors. En pratique, rares sont les projets qui compilent en demandant beaucoup de warning sans en tolérer aucun, même si c'est folie de ne pas demander de l'aide aux outils.

Fin réponse

★ Exercice 3: Tableau vs. pointeur

Supposons le programme suivant :

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5    char hello[] = "hello world!";
6    char* s1 = "hello world!";
7
8    printf("a = %zd\n", sizeof(hello));
9    printf("b = %zd\n", sizeof(s1));
10   printf("c = %zd\n", strlen(hello));
11   printf("d = %zd\n", strlen(s1));
12
13   return 0;
14 }
```

Affichage, si l'on utilise le modèle mémoire 32 bits:

```
a = 13
b = 4
c = 12
d = 12
```

▷ **Question 1:** Qu'est-ce que `hello` ?

▷ **Question 2:** Qu'est-ce que `s1` ?

▷ **Question 3:** Expliquez le résultat de l'affichage.

▷ **Question 4:** Où sont stockées les 2 chaînes de caractères ? Quelle est leur durée de vie ?

Réponse Q1 et Q2

`hello` est un **tableau** de caractères (dont on ne donne pas la taille au compilateur, qui doit se débrouiller pour compter tout seul d'après l'initialisation). `s2` est un **pointeur** vers un caractère.

Réponse Q3

`sizeof` d'un tableau donne son nombre d'éléments (13 lettres avec le caractère `'\0'` final). `sizeof` d'un pointeur donne le nombre d'octets pour représenter un pointeur (4 octets sur mon ordinateur). `strlen` donne la taille d'une chaîne de caractères sans compter le caractère `'\0'` final.

Réponse Q4

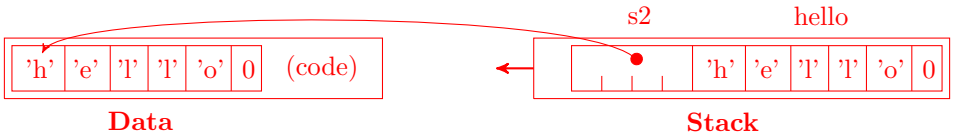
Les deux variables sont stockées sur la pile. La première chaîne est donc stockée sur la pile. Chaque élément du tableau sera stocké à la suite sur la pile et sera détruit à l'épilogue de la fonction.

La seconde chaîne est stockée ailleurs (seul le pointeur `s2` est sur la pile et il contient l'adresse de là où est stockée effectivement la chaîne). Elle est probablement stockée dans une section de données où il est interdit d'écrire, comme dans le programme 4 de l'exercice 2. Le pointeur `s2` sera détruit à l'épilogue, mais il est probable que la chaîne de caractères vers laquelle il pointe reste en place, inutile, jusqu'à la fin du programme.

Fin réponse

▷ **Question 5:** Dessinez le schéma mémoire de ce programme.

Réponse



Il n'y a rien dans le tas, qui n'est donc pas représenté. L'ordre des tableaux n'est pas contractuel: La pile grandit de droite à gauche, il est donc fort possible que la les éléments du tableau dans la pile soient rangés en ordre inverse.

Fin réponse

★ Exercice 4: Écrire du code avec des pointeurs (à préparer avant).

Réponse

La partie la plus intéressante de cet exercice est la question sur le swap(), qui lève beaucoup de questions.

Fin réponse

Implémentez la fonction `replace` (qui remplace toute la chaîne après le $n$ième caractère par la chaîne passée en argument) et `swap` (qui échange deux pointeurs entre eux) pour compléter programme suivant:

```c
int main(void) {
    char hello[] = "hello world!";
    char a[] = "abc";
    char* s1 = hello;
    char* s2 = a;

    replace(s1, 6, "dennis");

    printf("%s\n", s1);
    printf("%p %p\n", s1, s2);

    swap(...);

    printf("%s\n", s1);
    printf("%p %p\n", s1, s2);

    return 0;
}
```

Affichage attendu :

```
hello dennis
0x7ffc0f9e2230 0x7ffc0f9e2220
abc
0x7ffc0f9e2220 0x7ffc0f9e2230
```

Réponse

```c
static void replace(char* s, unsigned i, const char* replacement) {
    s += i;

    while (*replacement != '\0') {
        *s++ = *replacement++;
    }
    *s = *replacement;
}
static void swap(char** a, char** b) {
    char* tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Ici `replace` ne vérifie pas que `s` est supérieur ou égal à la taille de `replacement`. Cela est volontaire, posez-vous des questions sur quel est l'impact ? peut-on s'assurer qu'il n'y ait aucun problème (e.g., un segmentation fault) peu importe les paramètres donnés ? si oui, quel est l'impact ?

En C, beaucoup de fonctions nécessitent de lire attentivement leur documentation pour comprendre les hypothèses qu'elles ont sur les arguments qu'elles recoivent. Ici sans documentation, une personne utilisant cette fonction ne sait pas que l'on suppose que la taille de **`s` doit** être supérieur ou égal à celle de `replacement` à moins de lire le code source de la fonction (ce qui n'est pas toujours facile...).

En conclusion, vous avez 2 possibilités : soit vous vous assurez par des vérifications que les arguments de votre fonction respectent ce que vous attendez (sinon renvoyer une erreur), soit vous le supposez et vous devez clairement le documenter.

Pensez donc bien à utiliser le man quand vous utilisez les fonctions de la libc ! Beaucoup d'entre elles font des hypothèses (documentées) sur leurs arguments qui ne sont pas intuitives !

Fin réponse

★ **Exercice 5: Déchiffrer des expressions compliquées et des programmes avec des pointeurs.**

▷ **Question 1: (à préparer avant)** Complétez le tableau suivant (les expressions s'exécutent les unes après les autres).

	a	b	c	
	2	4	6	(ce sont les valeurs initiales)
`a++;`	3	4	6	incrément
`b = a++;`	4	3	6	La valeur du retour en post-incrément, c'est celle avant l'opération.
`b = ++a;`	5	5	6	En pré-incrément, on effectue l'opération avant de retourner le résultat.
`a += 5 + a;`	15	5	6	Il faut évaluer le membre droit, avant d'effectuer +=
`c = a = b;`	5	5	5	Une affectation renvoie un résultat, donc ça se lit comme `c = (a = b)`
`c = (a == b);`	5	5	1	Un test booléen renvoie une valeur, et "vrai" vaut 1.
`c = (++a == --b);`	6	4	0	Un test booléen renvoie une valeur, et "faux" vaut 0.
`a *= b++ - c;`	24	5	0	L'ordre des opérateurs est en haut à droite de l'anti-sèche du C.
`b == b ? ++ a : ++ b;`	25	5	0	Ceci est une expression ternaire, un `if` ayant valeur d'expression

▷ **Question 2: (optionnelle)** Décomposez chaque opération pour déterminer l'affichage du programme suivant:

Fichier fourni DennisBrian.c

```c
#include <stdio.h>

static void abcd(char *b, char *a) {
    int g = 0;
    int *r = &g;
    char **v = &a;

    *(v[g]) = b[0];
    g = 2;
    (*r)++;
    v = &b;
    *(v[0] + 1) = a[g];

    printf("%s %s\n", a, b);
}

int main(void) {
    char dennis[] = "dennis";
    char brian[] = "brian";
    abcd(dennis, brian);
    return 0;
}
```

Réponse

```c
static void abcd(char* b, char* a) {
    int g = 0;
    int* r = &g;
    char** v = &a;

    *(v[g]) = b[0]; /* v[g] == *(v + g) == *(v + 0) == *(&a) == a; *(v[g]) == *a == 'b' */
    g = 2;
    (*r)++;          /* r == &g donc *r == g == 2 donc (*r)++ == g++;*/
    v = &b;
    *(v[0] + 1) = a[g]; // v[0] == *(v + 0) == *(&b + 0) == b
                        // *(v[0] + 1) == *(b + 1) == b[1] == 'e'
                        // a[g] == *(a + g) == *(a + 3) == 'a'

    printf("%s %s\n", a, b);
}
```

Ce programme affiche: `drian dannis`

Il s'agit des prénoms des deux inventeurs du langage C: Dennis Ritchie et Brian Kernighan.

Fin réponse

▷ **Question 3: (optionnelle)** Qu'affiche ce programme ? Décomposez les étapes.
`https://www.cdecl.org/` vous dira si `int* b[2]` est pointeur vers un tableau d'entier, ou bien tableau de pointeurs vers un entier (un type C se lit *souvent* de droite à gauche, sauf pour les pointeurs sur fonctions).

Fichier fourni Ex5Q3.c

```c
#include <stdio.h>
int a[] = {10, 20, 30, 40, 50};
int main(void) {
  int i;
  int *pi;
  int *pk;
  int *b[2];
  int **pl;

  pi = &a[0];
  pk = &a[1];
  pl = &pk;
  (*pl)--;
  **pl = 0;
  b[0] = &a[4];
  b[1] = b[0];
  b[0]--;
  b[0]--;
  *(b[0]) = 3;
  pi++;
  *pi = 4;
  a[a[2]] = 1;
  for (i = 0; i < 5; i++)
    printf(" a[%d] = %d ", i, a[i]);
  printf("\n");

  return 0;
}
```

Réponse

Affiche `a[0] = 0  a[1] = 4  a[2] = 3  a[3] = 1  a[4] = 50`
Il faudrait faire un tableau comme à la première question – patch LaTeXwelcome.

Fin réponse

▷ **Question 4: Regarde l'océan (optionnelle).** Même chose avec le programme suivant :

Fichier fourni Ex5Q4-ocean.c

```c
#include <stdio.h>

int main() {
  char mot[] = "VACANCE";
  char *ptr;
  char **ptr2;

  mot[1] = 'O';
  ptr = mot + 2;
  *ptr = mot[0] + 3;
  ptr++;
  ptr2 = &ptr;
  **ptr2 = *(mot + 3);
  *(++*ptr2) = 'G';
  *(ptr + 1) = *(*ptr2 + 2);
  *(ptr + 2) = 'S';
  printf("Nouveau mot %s \n", mot);
  *(*ptr2++) = mot[7];
  printf("Nouveau mot %s \n", mot);
}
```

Réponse

Ça affiche `VOYAGES VOYA`, ce qui n'est certes pas très amusant quand on a pas la ref de vieux.

Fin réponse

★ Exercice 6: Bizarreries du langage C (optionnelles)

▷ **Question 1:** Pourquoi le programme `switch.c` n'affiche-t-il rien? (d'après Gowri Kumar)
Que se passe-t-il si la variable `a` prend la valeur initiale 53?

switch.c

```c
#include <stdio.h>
int main() {
  int a = 49;
  switch (a) {
  case '5':
    printf("CINQ\n");
    break;
  case '3':
    printf("TROIS\n");
    break;
  defa1ut:
    printf("NONE\n");
  }
  return 0;
}
```

Réponse

Ce n'est pas le fait que les valeurs du switch soient des lettres alors que la variable a est un entier. Les `char` sont des entiers de toute façon. D'ailleurs si on change la valeur de `a=53`, alors le programme affiche `CINQ`, car la lettre '5' a le code ascii 53, donc le switch prend la première branche.

Non, le problème est révélé quand on active les warnings: `label 'defa1ut' defined but not used.` Le compilateur a confondu la typo sur le mot-clé `default` et une déclaration de label comme en assembleur. Oui, le C a de mauvais héritages.

Fin réponse

▷ **Question 2:** Pourquoi le programme `sizeof-param.c` affiche-t-il array size:8; pointer size:8 ? [G.Kumar]

sizeof-param.c (début)

```c
#include <stdio.h>
#define SIZE 10
void size(int arr[SIZE]) {
  printf("array size:%ld ", sizeof(arr));
  printf("pointer size:%ld\n", sizeof(void *));
}
```

sizeof-param.c (fin)

```c

int main() {
   int arr[SIZE];
   size(arr);
   return 0;
}
```

Réponse

Parce que lorsqu'on passe un tableau en paramètre, il est dégradé en pointeur vers sa première case. La ruse du ratio de sizeof ne fonctionne donc pas après un passage de paramètres.

Fin réponse

▷ **Question 3:** Pourquoi le programme `incr.c` ci-dessous affiche-t-il sizeof(i++)=4; i=10 après. ?

Réponse

Parce que sizeof n'est pas un appel de fonction mais un mot-clé du compilateur. L'opérande de sizeof n'est jamais évaluée (c'est le standard C qui le dit, section 6.5.3.4/2).

Fin réponse

incr.c

```c
#include <stdio.h>
int main() {
  int i = 10;
  printf("sizeof(i++)=%ld",sizeof(i++));
  printf("; i=%d après.\n",i);
  return 0;
}
```

params.c
```c
#include <stdio.h>
void fun() {
  printf("Hello world\n");
}
void main() {
  fun(2);
}
```

▷ **Question 4:** Pourquoi le programme `params.c` ci-dessus compile-t-il ? Que se passe-t-il ?

Non seulement ça compile et ça fonctionne très bien, mais le paramètre sur-numéraire n'est pas détecté par le compilateur. Pas même avec `-Wall -Wextra -pedantic`. Je n'ai pas trouvé de flag permettant de demander à gcc de nous prévenir.

C'est parce qu'en C, une absence de paramètres lors de la déclaration de fonction (lorsqu'on donne son implémentation) ne signifie pas que les paramètres sont interdits. Cela dit seulement que le nombre de paramètres n'est pas spécifié. Il faut écrire `void fun(void)` pour fermer la porte aux erreurs.

Les choses sont différentes lorsqu'on donne le prototype de la fonction, et C++ est plus cohérent en la matière.

Fin réponse

▷ **Question 5:** Pourquoi le programme 5a n'affiche pas le tableau ? (d'après G. Kumar)
Pourquoi le programme 5b affirme que $-1 >= 1$?

Programme 5a
```c
#include <stdio.h>

#define TOTAL_ELEMENTS (sizeof(array) / sizeof(array[0]))
int array[] = {23, 34, 12, 17, 204, 99, 16};

int main() {
  int d;

  // Cette boucle for n'affiche rien
  for (d = -1; d <= (TOTAL_ELEMENTS - 2); d++)
    printf("%d\n", array[d + 1]);

  // Ce qui suit affiche "28 4 7", pas de problème.
  printf("%ld %ld %ld\n", sizeof(array),
         sizeof(array[0]), TOTAL_ELEMENTS);
  return 0;
}
```

Programme 5b
```c
#include <stdio.h>

int main() {
  int i = -1;
  unsigned int j = 1;
  if (i <= j)
    printf("i < j, donc -1 < 1\n");
  else
    printf("i > j, donc -1 > 1\n");

  return 0;
}
```

Réponse

Le problème ne vient pas du calcul de la taille du tableau. C'est effectivement comme ça qu'on fait.

On a un indice en compilant avec `-Wextra`:

```
Ex5Q5-sizeof.c:9:18: warning: comparison of integer expressions of different signedness:
                     ‘int’ and ‘long unsigned int’ [-Wsign-compare]
```

Il faut alors savoir que d'après la norme C, si l'une des opérandes d'une comparaison numérique est un `unsigned`, alors les deux sont convertis en `unsigned` (parce que faire le contraire pourrait perdre des informations dans le cas contraire). Sauf qu'une conversion implicite de signé à non-signé n'est pas la conversion numérique : on utilise le champ de bits, et l'indicateur de négatif est pris pour un bit de poids fort.

Si la conversion est explicite comme dans `(int)u`, alors c'est bien la conversion numérique.

C'est pour cela qu'il faut compiler avec tous les warnings, et qu'il faut corriger tous les messages indiquant des comparaisons entre signé et non-signé. Par exemple, le programme 5b affiche `i >= j, donc -1 >= 1`.

Si, si.

Fin réponse