Devoir surveillé #1

Vendredi 2025-11-07 ; durée : trois heures

Petits exercices

Exercice 1. *Lecture de types*

Soit l'extrait de programme C suivant :

```c
struct intpair
{
    int a;
    int b;
};

struct intpair fun(int a, int b)
{
    int x;
    unsigned *p;
    int u[10];
    bool v;
    // ...
}
```

1. Donnez la signature de la fonction y apparaissant, et expliquez ce qu'elle signifie (seulement en tant que signature ; on ne cherche pas particulièrement à deviner ce que fait réellement la fonction).
2. Pour chacune des variables apparaissant dans cette fonction, donnez son **type C** et expliquez brièvement ce en quoi il consiste.

Exercice 2. *Exécution de programme*

Soit le code source C suivant :

```c
#include <stdlib.h>
#include <stdio.h>

void fun(unsigned n) {
    for (unsigned i = 0; i < n; i = i + 2) {
        printf("%d ", i);
    }
    printf("\n");
}

int main(void) {
    fun(10);
    return EXIT_SUCCESS;
}
```

1. Donnez une suite possible de commandes permettant de le compiler et d'exécuter le programme résultant depuis un terminal UNIX (avec un environnement similaire à ceux disponibles en TP). Pour cela, on supposera que le code source est contenu dans un fichier `pe.c` se trouvant dans le même répertoire que le répertoire de travail du terminal.
2. Sans justification, que produit l'exécution de ce programme ?

Exercice 3. *Trouvez les bugs !*

On constate à l'exécution que la boucle suivante ne termine pas :

```c
for (unsigned i = n; i >= 0; i--) {
    n = n + i;
}
```

1. Expliquez pourquoi.

On se donne la fonction suivante, dont on souhaite qu'elle renvoie `true` si `x` est un élément de `a`, et `false` sinon :

```c
bool search(size_t n, int a[n], int x) {
    for (size_t i = 0; i < n; i++) {
        if (a[i] == x) {
            return true;
        }
        else {
            return false;
        }
    }
}
```

2. Expliquez en quoi elle n'est pas correcte pour ses spécifications.
3. Proposez une correction (on ne demande pas de justifier que la fonction corrigée soit correcte).
4. Quelle erreur est causée par l'extrait de code suivant :

```c
int *p = NULL;
*p = 12;
```

5. Quelle erreur est causée par l'extrait de code suivant :

```c
int a[10];
for (size_t i = 0; i <= 10; i++) {
    a[i] = i;
}
```

Exercice 4. *Terminaison 1*

On se donne la fonction `fun` suivante :

```c
unsigned fun(unsigned x) {
    unsigned r = 1;
    while (r < x) {
        r = 2 * r;
    }
    return r;
}
```

On suppose dans un premier temps que le type `unsigned` permet de représenter l'ensemble des entiers naturels.

1. Prouvez la terminaison de cette fonction.

On considère maintenant le « vrai » type `unsigned`.

2. Expliquez pourquoi la terminaison de la fonction `fun` n'est plus garantie.

Pour les questions suivantes on pourra admettre l'invariant (global) suivant : la valeur de `r` est une puissance de deux ou nulle.

3. Donnez un exemple d'argument pour lequel `fun` ne termine pas (on ne demande pas nécessairement une valeur numérique exacte ; une description suffisamment précise suffit).
4. Proposez une *précondition* sur l'argument de cette fonction qui, quand elle est satisfaite, garantit sa terminaison. On cherchera à avoir une précondition *la moins restrictive possible*, mais l'on pourra se contenter d'une argumentation sommaire.

Exercice 5. *Terminaison 2*

On se donne la fonction `fun` suivante :

```c
unsigned fun(unsigned a, unsigned b) {
    unsigned r = 0;
    while (a > 0 || b > 0) {
        if (a > b) {
            a = a - 1;
        }
        else {
            b = b - 1;
        }
        r = r + 1;
    }
    return r;
}
```

1. Prouvez sa terminaison.

Exercice 6. *Popcount*

Soit un entier non signé `x` de type `uint32_t`, on définit son *population count* comme le nombre de bits à 1 dans son écriture en base deux.

1. Écrivez une fonction C de signature :
   `uint32_t popcount(uint32_t x)`
   qui calcule et renvoie le *population count* de son argument.
2. Proposez un jeu de 3 tests pour cette fonction.

Exercice 7. *Somme de tableaux*

1. Écrivez une fonction `sum` de signature :
   `unsigned *sum(size_t n, unsigned a[n], unsigned b[n])`
   qui alloue une zone mémoire de durée de stockage « alloué » permettant de stocker `n` entiers `unsigned` consécutifs (on pourra supposer `n` non nul), l'initialise de façon à ce que le $i$ème de ces entiers soit égal à la somme des éléments d'indice i de `a` et `b`, et renvoie l'adresse du début de la zone.
   On pourra considérer que l'allocation mémoire réussit toujours.

Problèmes

Exercice 8. *Optimalité de la recherche dichotomique*

Le but de cet exercice est de prouver dans un certain sens que la recherche par dichotomie d'un élément dans un tableau trié est optimale (la plus efficace possible), à constantes près.
On s'intéresse donc à des algorithmes résolvant le problème suivant : soit en entrée un tableau a trié de n entiers et un entier x, on veut renvoyer :

— un entier positif $i \in [\![0, n-1]\!]$ tel que $a[i] = x$ si un tel entier i existe
— -1 sinon

1. Expliquez le principe d'une recherche par dichotomie, et comment elle permet de résoudre le problème ci-dessus.
2. Justifiez brièvement que le coût d'une recherche par dichotomie est un $O(\log n)$ dans le pire cas.

On considère maintenant des algorithmes (par exemple écrits en C) qui ne peuvent pas directement accéder aux éléments de a, et ne peuvent apprendre de l'information à leur sujet qu'à travers une fonction de signature :
`int cmp(size_t i, int x)`
qui renvoie respectivement `-1`, `0` ou `1` si `a[i]` est strictement inférieur, égal ou strictement supérieur à `x`. On interdit également à l'algorithme de faire appel à toute autre fonction. Un algorithme résolvant le problème de recherche a alors par exemple une signature :
`int search(size_t n, int x)`

(où l'on suppose `n` représentable par un `int`).
On donne ci-dessous un exemple d'un tel algorithme résolvant le problème par une recherche linéaire :

```
int search(size_t n, int x) {
    for (size_t i = 0; i < n; i++) {
        if (cmp(i, x) == 0) {
            return i;
        }
    }
    return -1;
}
```

Le but va maintenant être de montrer que tout algorithme résolvant correctement le problème doit effectuer au moins K appels à la fonction `cmp` dans le pire cas, avec $K = \Omega(\log n)$ (autrement dit, K est asymptotiquement supérieur ou égal à $\log n$, à constantes près) ; ceci permettra immédiatement de conclure que le coût pire cas de l'algorithme lui même est un $\Omega(\log n)$.
Dans tout ce qui suit, on note $\mathbb{A}$ un algorithme résolvant (totalement) correctement le problème de recherche, et n et x respectivement la longueur du tableau dans laquelle la recherche s'effectue et l'élément recherché.

3. Montrez que pour toute valeur x fixée, $\mathbb{A}$ peut renvoyer exactement $n+1$ réponses différentes, en considérant l'ensemble des tableaux de n entiers (par exemple de type `int`) sur lesquels il peut s'exécuter.

Soit une exécution (qui termine) de $\mathbb{A}$, on appelle *transcript* la suite finie des valeurs renvoyées par les appels successifs à `cmp` (qui dépend en principe de n, x, et les valeurs contenues dans le tableau).

4. Montrez qu'on peut majorer le nombre de transcripts distincts de longueur positive ou nulle inférieure ou égale à c par 3^{c+1}.

On admet maintenant que pour des valeurs de n et x fixées, la réponse renvoyée par $\mathbb{A}$ dépend *uniquement* du transcript de son exécution (c'est à dire, des valeurs contenues dans le tableau). (On peut justifier cela par le fait qu'une fois les arguments de $\mathbb{A}$ fixés, la seule variabilité dans son exécution peut provenir des valeurs renvoyées par les fonctions appelées par $\mathbb{A}$, et que ces appels ne peuvent être qu'à `cmp`.)

5. Conclure.

Exercice 9. *Implémentation jouet de la fonction de hachage de mot de passe* `scrypt`

Dans tout cet exercice, on suppose disposer d'une fonction `die` qui prend en argument une chaîne de caractère, affiche celle-ci sur la sortie standard, et termine l'exécution du programme avec un code d'erreur. Une implémentation possible de cette fonction est :

```
void die(char *msg) {
    fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}
```

et un exemple d'utilisation :

```
die("erreur à l'exécution");
```

Le but de cet exercice est d'implémenter une version jouet de la *fonction de hachage de mot de passe* `scrypt`. On définit celle-ci de la façon suivante (sans pour l'instant préciser les types exacts des objets manipulés) :
soit H une *fonction de hachage* prenant pour argument un mot de passe m et une clef k et renvoyant un entier de 64 bits, et r un entier strictement positif (potentiellement grand) inférieur à 2^{64}, on définit :

- $h_0 = H(k, m)$, puis $h_i = H(k, h_{i-1})$ pour $i \in [\![1, r-1]\!]$.
- $b_0 = H(k, h_{r-1})$, puis $b_i = H(k, b_{i-1} \boxplus h_{b_{i-1} \bmod r}$ pour $i \in [\![1, r-1]\!]$, (où « $\boxplus$ » désigne l'addition « modulo 2^{64}, et « $a \mod b$ » le reste positif de la division de a par b, tous deux positifs).
- La valeur de `scrypt` pour ces arguments et cette fonction de hachage est définie comme l'entier de 64 bits représentant b_{r-1}.

Pour la suite, on suppose les mots de passe représentés comme des tableaux d'entiers `uint64_t`, et la fonction de hachage H implémentée par une fonction de signature :

```
uint64_t hash(uint64_t k, size_t n, const uint64_t m[n])
```

Le qualifieur de type `const` indique le fait que cette fonction ne modifie pas les éléments de son argument `m` (ce que l'on pourra donc supposer chaque fois que l'on utilise la fonction).

1. Écrivez un extrait de code C qui appelle `hash` avec un argument `k` quelconque et comme argument `m` un tableau `a` (supposé déjà déclaré) de longueur `100`.
2. Expliquez ce que fait l'extrait de code suivant :
   ```
   uint64_t a = 123456;
   uint64_t h = hash(0, 1, &a);
   ```
3. Écrivez une fonction `scrypt_1` de signature :
   ```
   uint64_t scrypt_1(uint64_t k, size_t n, uint64_t m[n], uint64_t r)
   ```
   qui calcule et renvoie le résultat de `scrypt` telle que définie ci-dessus, où `k` correspond à k, `m` (de longueur `n`) à m, et `r` à r.
   Cette fonction ne devra calculer chaque valeur h_i qu'une unique fois au cours de son exécution, et pourra allouer de la mémoire avec `malloc`.
   Prenez garde à correctement traiter les éventuelles erreurs pouvant avoir lieu à l'exécution, et à éviter toute fuite mémoire.
4. Écrivez une fonction `scrypt_2` de signature :
   ```
   uint64_t scrypt_2(uint64_t k, size_t n, uint64_t m[n], uint64_t r)
   ```
   qui calcule et renvoie le résultat de `scrypt` telle que définie ci-dessus, où `k` correspond à k, `m` (de longueur `n`) à m, et `r` à r.
   Cette fonction ne devra pas utiliser `malloc`, mais pourra calculer plusieurs fois chaque valeur h_i au cours de son exécution.
5. Pour chacune de vos fonctions `scrypt_1` et `scrypt_2`, donnez (asymptotiquement et à constante près, exprimé sous forme de O) :
   - le nombre F d'appels à `hash` ;
   - le nombre M d'accès à de la mémoire allouée par `malloc` (en ignorant les éventuels accès mémoire fait par d'autres fonctions, comme `hash`) ;

   en fonction de r.
6. Sans l'implémenter, proposez un *compromis temps-mémoire* qui permette d'implémenter `scrypt` de façon à avoir M (défini comme à la question précédente) pouvant prendre n'importe quelle valeur strictement positive, et qui est telle que $FM = O(r^2)$ (avec F également défini comme à la question précédente).