Cours d'introduction à l'informatique
Énoncés et solutions de l'examen de juin 2019

Énoncés

1. (a) Écrire une fonction C prenant en arguments deux entiers strictement positifs n et f, et calculant la multiplicité du facteur f dans n, c'est-à-dire le plus grand nombre entier m tel que f^m divise n. Par exemple, si $n = 324$ et $f = 9$, on a $m = 2$ car $324 = 9^2.4$.

   (b) Par la méthode des invariants, démontrer que la valeur retournée par cette fonction est correcte.

2. En langage C :

   (a) Qu'est-ce qu'un prototype, et à quoi un prototype sert-il ?

   (b) Quelle est la valeur affichée lors de l'exécution du fragment de code suivant ? (Justifier votre réponse.)

```
int t[2] = { 1, 2 };
int *a, **b;

a = t + *t;
b = &a;

printf("%d\n", *--*b);
```

3. (a) Décrire, le plus simplement possible, l'opération effectuée par la fonction C suivante.

```
unsigned f(unsigned v)
{
  int n;

  if (!v)
    return 0;

  n = v % 10 == 9 ? 1 : 0;

  return f(v / 10) + n;
}
```

   (b) Quelle est la complexité en temps de cette fonction ?

(c) Écrire une fonction C réalisant exactement la même opération, mais sans effectuer d'appel récursif.

4. (a) Écrire un fragment de code C définissant un type structuré capable de retenir le nom et le prénom d'une personne, tous deux représentés sur au plus 20 caractères.

(b) Écrire une fonction C prenant en argument des pointeurs vers deux instances du type obtenu à la question précédente, et retournant une valeur booléenne qui indique si les deux personnes concernées possèdent le même nom et le même prénom. La comparaison doit tenir compte des caractères non alphabétiques, mais pas de la casse (majuscule ou minuscule) des lettres ; par exemple, les noms "le petit" et "Le Petit" sont considérés égaux, mais différents de "Lepetit". Il est permis de définir des fonctions auxiliaires de votre choix, mais pas de faire appel à des fonctions issues d'une bibliothèque.

Exemples de solutions

1. (a) Une façon simple de résoudre le problème consiste à diviser répétitivement n par f tant que cela est possible, et à compter le nombre de fois où cette opération est effectuée. On obtient le code suivant :

```
unsigned multiplicite_facteur(unsigned n, unsigned f)
{
  unsigned  m;

  for (m = 0; !(n % f); m++)
    n /= f;

  return m;
}
```

(b) On ne demande pas de prouver que la fonction termine toujours son exécution, mais de démontrer que lorsque cette exécution se termine, la valeur retournée est correcte. Cela revient à prouver le triplet

$$\{\texttt{n} = n_0 > 0,\ \texttt{f} > 0\}$$

```
for (m = 0; !(n % f); m++)
  n /= f;
```

$$\{\texttt{m} = \text{multiplicité de } \texttt{f} \text{ dans } n_0\},$$

où n_0 dénote la valeur initiale de `n`. En développant la boucle `for`, cela équivaut à démontrer

$$\{\texttt{n} = n_0 > 0,\ \texttt{f} > 0,\ \texttt{m} = 0\}$$

```
while (!(n % f))
  {
    n /= f;
    m++;
  }
```

$$\{\texttt{m} = \text{multiplicité de } \texttt{f} \text{ dans } n_0\}$$

Cherchons un invariant de boucle I. Celui-ci doit satisfaire les trois propriétés suivantes :

— I doit être impliqué par la précondition ($\texttt{n} = n_0 > 0$ et $\texttt{f} > 0$ et $\texttt{m} = 0$).

— Si I est vrai avant une itération de la boucle `while`, alors I doit également être vrai à la fin de cette itération.

— Après la dernière itération de la boucle, I doit impliquer la postcondition ($\texttt{m} =$ multiplicité de $\texttt{f}$ dans n_0).

Ces trois propriétés sont vraies pour l'invariant

$$I : \texttt{n} > 0 \text{ et } \texttt{f} > 0 \text{ et } n_0 = \texttt{n}.\texttt{f}^{\texttt{m}}.$$

En effet :

— Initialement, on a ($\texttt{n} = n_0 > 0$ et $\texttt{f} > 0$ et $\texttt{m} = 0$) $\Longrightarrow I$.

— D'une itération à l'autre de la boucle, on a le triplet

$$\{I,\ \texttt{n} \text{ est divisible par } \texttt{f}\}$$

```
n /= f;
m++;
```

$$\{I\},$$

qui est bien valide. En effet, si $\texttt{n}'$ et $\texttt{m}'$ dénotent respectivement les valeurs de `n` et `m` après l'exécution du fragment de code, on a

$$\texttt{n}' = \frac{\texttt{n}}{\texttt{f}} \quad \text{et} \quad \texttt{m}' = \texttt{m} + 1,$$

donc

$$(I \text{ et } \texttt{n} \text{ est divisible par } \texttt{f})$$

implique

$$(\texttt{n}' > 0 \text{ et } \texttt{f} > 0 \text{ et } n_0 = \texttt{n}.\texttt{f}^{\texttt{m}} = \texttt{n}'.\texttt{f}^{\texttt{m}'}).$$

— En sortie de boucle, on a

$$(I \text{ et } \texttt{n} \text{ n'est pas divisible par } \texttt{f})$$

qui implique bien

$$(\texttt{m} = \text{multiplicité de } \texttt{f} \text{ dans } n_0),$$

car si $n_0 = \texttt{n}.\texttt{f}^{\texttt{m}}$ et `n` n'est pas divisible par `f`, alors `m` est la multiplicité de `f` dans n_0 (par définition de cette multiplicité).

2. (a) Un prototype est une déclaration qui précise le nom d'une fonction, le type de ses paramètres, et le type de la valeur qu'elle retourne. Il permet au compilateur de vérifier que les appels à cette fonction et la définition de celle-ci sont conformes à cette déclaration, et d'effectuer les conversions de type nécessaires.

   (b) — L'évaluation de `*t` retourne la valeur de la première case du tableau pointé par `t`, c'est-à-dire 1.

   — La première instruction place donc dans `a` un pointeur égal à `t + 1`, c'est-à-dire un pointeur vers la deuxième case du tableau `t`.

   — L'instruction `b = &a;` place dans `b` un pointeur vers la variable `a`.

   — L'expression `--*b` décrémente d'une unité la valeur pointée par `b`, c'est-à-dire le contenu de la variable `a`. Cette expression s'évalue donc en un pointeur vers la première case du tableau `t`.

   — L'opérateur `*` appliqué à cette expression retourne donc le contenu de la première case du tableau pointé par `t`, c'est-à-dire 1.

   — Le programme affiche donc "1" (suivi d'un saut à la ligne).

3. (a) Cette fonction compte le nombre de chiffres égaux à 9 dans l'écriture décimale du nombre `v` qui lui est passé en argument.

   (b) Pour un nombre $\texttt{v} > 0$ donné, l'évaluation de `f(v)` provoque $\lfloor \log_{10} \texttt{v} \rfloor + 1$ appels à la fonction `f`, où $\lfloor x \rfloor$ dénote le plus grand entier inférieur ou

égal à x. Chacun de ces appels exécute un nombre borné d'instructions. La complexité en temps de cette fonction vaut donc $O(\lfloor \log_{10} \mathtt{v} \rfloor + 1) = O(\log \mathtt{v})$.

(c)
```
unsigned f(unsigned v)
{
  int n;

  for (n = 0; v; v /= 10)
    if (v % 10 == 9)
      n++;

  return n;
}
```

4. (a)
```
typedef struct
{
  char nom[21];
  char prenom[21];
} personne;
```

(b)
```
int chaines_egales(char *s1, char *s2)
{
  for (; *s1 || *s2; s1++, s2++)
    {
      if (*s1 == *s2)
        continue;

      if (*s1 >= 'A' && *s1 <= 'Z' &&
          *s1 + 'a' - 'A' == *s2)
        continue;

      if (*s2 >= 'A' && *s2 <= 'Z' &&
          *s2 + 'a' - 'A' == *s1)
        continue;

      return 0;
    }

  return 1;
}
```

```
int personnes_egales(personne *p1, personne *p2)
{
  return chaines_egales(p1 -> nom, p2 -> nom) &&
         chaines_egales(p1 -> prenom, p2 -> prenom);
}
```