UGA - UFR IM^{2}AG **Master 2 CCI**

Programmation et Langages 1

Examen du 30 novembre 2018 (session 1)

Duré : 2h – **Documents :** autorisés – Barême indicatif (sur 20 points) – 3 exercices indépendants

Exercice 1 (∼ 5 points)

Pour représenter un nombre entier n de la forme r^p, une solution possible consiste à utiliser un couple d'entiers (r, p), où r est appelé *racine* et p est appelé *exposant*. Ainsi, le nombre 3^7 peut être représenté par le couple $(3, 7)$. On supposera dans la suite que la racine et l'exposant sont des entiers positifs ou nuls.

Q1. Définissez en C le type `Puissance` comme un couple contenant deux champs entiers positifs ou nuls nommés `racine` et `exposant`.

Q2. On rappelle que $r^{p1} \times r^{p2} = r^{p1+p2}$ et $r^{p1^{p2}} = r^{p1.p2}$.
Ecrivez le corps des deux fonctions suivantes :

```
Puissance produitPuiss (Puissance n1 ; Puissance n2) ;
// pre-condition : n1 et n2 ont meme racine
// produitPuiss(n1, n2) renvoie le produit n1 x n2

Puissance expPuiss (Puissance n1 ;  int p2) ;
// expPuis(n1, p2) renvoie n1^p2, c'est-a-dire  n1 a la puissance p2
```

Q3. Ecrivez le corps de la fonction suivante :

```
int valeurpuiss (puissance n) ;
// si n = (r,p) alors valeurPuiss(n) renvoie l'entier r^p
// exemple : si n = (2,4) alors valeur(n) = 16
```

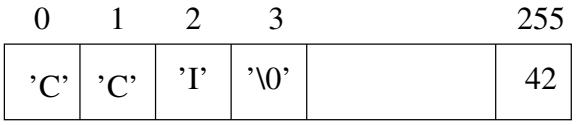
Indication : il n'y a pas d'opérateur "puissance" en C défini sur les entiers. On pourra donc, par exemple, s'inspirer de l'algorithme suivant qui calcule dans une variable `resultat` la valeur r^p :

resultat ← 1
pour i parcourant 1 .. p
    resultat ← resultat × r

Q4. D'après vous quel peut-être l'intérêt d'utiliser ce type `Puissance` dans un programme C (au lieu d'utiliser par exemple le type `int`) ?

(suite page suivante)

FIGURE 1 – Bloc pour le message `CCI` de checksum 42



Exercice 2 (∼ 3 points)

On dispose d'un programme constitué de plusieurs fichiers sources :
- `types.h` contient des définitions de type ;
- (`saisie.h`, `saisie.c`) fournissent les en-têtes et corps de fonctions de saisie ;
- (`calcul.h`, `calcul.c`) fournissent les en-têtes et corps de fonctions de calcul ;
- `main.c` contient le programme principal.

Les dépendances entre ces fichiers sont les suivantes :
- `main.c` utilise des fonctions de saisie et de calcul, il inclue donc les fichiers correspondants ;
- `types.h` est inclus par `main.c`, `saisie.c` et `calcul.c`.

Q1. Ecrivez un `Makefile` permettant de compiler ce programme, par **compilation séparée**, en un exécutable de nom `main`.

Q2. Après une première compilation (sans erreurs), l'utilisateur modifie `saisie.c`.
Quelles commandes seront exécutées lors du prochain appel à la commande `make` ?

Problème (∼ 12 points)

On souhaite transmettre des messages sur un réseau non fiable, c'est-à-dire un réseau sur lequel le contenu des messages risque d'être modifié. Pour détecter de telles erreurs de transmission, une solution possible consiste à associer à chaque message m un "code de contrôle" c (ou *checksum*, en anglais). On transmet donc un couple (m,c), où c dépend de m. De son coté, lorsque le récepteur reçoit un couple (m', c'), il peut calculer à son tour le checksum de m' et vérifier qu'il est bien égal à c'. Si ce n'est pas le cas il sait que la transmission a été perturbée (m' est incorrect ou c' est incorrect), et il peut demander qu'on lui retransmette à nouveau le message. Dans le cas contraire il peut supposer que la transmission s'est déroulée correctement.

Partie 1

On suppose tout d'abord que le message m à transmettre est une chaîne de caractères, terminée par le caractère `'\0'`, et contenant au plus 255 caractères. On suppose également que le checksum c est un entier compris en 0 et 255 (il peut donc être représenté par une valeur du type `unsigned char` en langage C).
Dans la suite nous supposerons donc que le couple (m,c) est mémorisé dans un tableau b de type `Block` tel que
- $b(0..l) = m$, avec l la longueur du message m (incluant le caractère `'\0'`) ;
- $b(255) = c$.

La Figure 1 donne le bloc obtenu pour le message `"CCI"` dont le checksum vaut (par exemple !) 42.

On définit donc en C le type `Block` permettant de représenter un message et son checksum :

```
#define N 256
typedef unsigned char Block[N] ;
```

On suppose également que l'on dispose d'une fonction `checksum()` qui calcule le checksum d'un message :

```
unsigned char checksum (unsigned char *m) ;
// m est une chaine de caracteres terminee par '\0'
//  checksum(m) renvoie le checksum du message m
```

Q1. Ecrire en C la fonction suivante, qui initialise un `Block b` en lisant une chaîne de caractères au clavier, en la mémorisant dans b et en lui ajoutant son checksum. Le message sera lu caractère par caractère (avec la fonction `scanf`), le dernier caractère fourni au clavier étant `'\0'`. On supposera que la chaîne entrée au clavier ne comporte pas plus de 255 caractères.

```
void makeBlock (Block b) ;
// e. i. : quelconque
// e. f. : b contient une chaine de caracteres lue au clavier (terminee par '\0') et son checksum
```

Q2. Ecrire en C la fonction suivante qui vérifie qu'un bloc est *correct*, c'est-à-dire qu'il contient un message m et le checksum c qui lui correspond.

```
int checkBlock (Block b) ;
// checkBlock(b) est vrai ssi b contient une chaine de caracteres (terminee par '\0')
// et son checksum
```

Q3. Ecrire la fonction `checksum (unsigned char *m)`, qui renvoie le checksum de `m`, en vous basant sur la solution suivante : le checksum de `m` est calculé en faisant la somme, modulo 255, des codes Ascii de tous les caractères présents dans m. On rappelle qu'en C un caractère est représenté par son code Ascii, on peut donc directement additionner 2 valeurs de type `unsigned char`.

Q4. Avec la fonction `checksum` proposée à la question **Q3**, est-il possible d'avoir 2 messages différents ayant même checksum ? Que peut-on en conclure ? Justifiez votre réponse ...(en quelques lignes).

Partie 2

On souhaite maintenant transmettre des messages de plus de 255 caractères en conservant le même mode de transmission. On découpe ainsi chaque message en une séquence de chaînes contenant chacune au plus 255 caractères, et on associe son checksum à chacune de ses chaînes. Le type `Message` est donc une **liste chaînée** de `Block`.

Q5. Définissez en C le type `Message`.

Q6. Ecrire en C une fonction `makeMsg` qui renvoie un `Message m` construit à partir d'une chaîne de caractères $s = s_1.s_2 \dots s_n$ lue au clavier. Le message m sera donc une liste chaînée de n blocs b_i telle que chaque bloc b_i contient :

- la sous-chaîne s_i de `s` d'au plus 254 caractères, terminée par `'\0'`
- le checksum de s_i.

La chaîne initiale s sera lue caractère par caractère (avec la fonction `scanf`), le dernier caractère fourni au clavier étant `'\0'`.

Q7. Ecrire en C une fonction qui vérifie si un `Message m` fourni en paramètre est *correct*, c'est-à-dire qui renvoie vrai si et seulement si chaque bloc constituant ce message `m` contient un checksum correct.

Q8. Ecrire en C une action qui prend un `Message m` en paramètre "donnée-résultat" et supprime les blocs de `m` qui n'ont pas un checksum correct.