Examen de Programmation C/Java

– Examen (**1**) –

Vendredi 9 décembre, 2016
Durée: 3 heures

Résumé

Ce sujet comprend un problème à traiter en C et en Java. À l'issue de l'examen vous devrez envoyer votre travail par e-mail dans une archive contenant le code source des programmes à l'adresse : `emmanuel.fleury@u-bordeaux.fr`.

L'archive devra avoir la forme suivante :

```
NOM_Prenom-examen/
    +-- poquemon/
    +-- java/
```


1 Programmation C : Poquemon (12 points)

1.1 Mise en place du projet

Sachant qu'il n'y a aucune dépendance à une bibliothèque particulière, le but de cette section est de réaliser un build-system avec l'outil `make` qui permette de compiler l'exécutable demandé.

Questions

1. Mettre en place la structure complète des sources du programme comme suit :

```
NOM_Prenom-examen/
    +-- poquemon/
          +-- poquemon.c
          `-- Makefile
```

2. Créez le fichier `Makefile` et écrivez les cibles suivantes :
   — `all` : Lance la compilation de l'exécutable `poquemon` ;
   — `poquemon` : Compile le fichier source en un exécutable ;
   — `clean` : Nettoie le répertoire de tous les fichiers superflus et des fichiers crées par la compilation.
3. Au sein de `Makefile` utilisez (et positionnez) correctement les variables classiques, c'est à dire : `CFLAGS`, `CPPFLAGS` et `LDFLAGS`. Ainsi que la cible spéciale `.PHONY`.

1.2 Problème principal

Cette année les elfes du pôle Nord ont encore fait une bêtise ! Ils voulaient faire une surprise au Père Noël en déballant eux-mêmes une machine qui permet d'imprimer et d'empaqueter des decks de cartes "*Poquémon*" à collectionner (un jeu qui fait fureur). Mais ils ont fait une erreur dans le montage et la machine ne trie pas correctement les paquets à la sortie de l'impression. Les cartes dans une pile ne sont donc pas systématiquement avec la face qu'il faut sur le dessus. En général, les piles sont assez désordonnées et il faut vite les trier avant que le tapis sur lequel elles sont ne les amène à l'emballage.

Les elfes ne savent pas du tout comment réparer la machine, alors ils ont décidé qu'ils trieraient les piles de cartes "au vol" pendant le trajet en tapis roulant entre le module d'impression et le module d'empaquetage. Mais, comme ce trajet est court, il faut faire vite. Heureusement, grâce à leurs dons surnaturels, ils savent exactement comment se présente un pile de cartes au premier regard.

Nous représenterons chaque carte dans une pile par des '`+`' (carte dans le bon sens) et des '`-`' (carte à l'envers). Ainsi, une pile de 5 cartes peut donner la représentation suivante : `++---` (à gauche, le haut de la pile, avec deux cartes dans le bon sens, puis trois cartes dans le mauvais sens).

La seule opération que les elfes peuvent faire c'est de se saisir d'un tas de `n` cartes en partant du sommet de la pile et les retourner. Ils peuvent éventuellement prendre toute la pile et la retourner s'ils veulent. Ainsi, si l'on prend la pile `++---`, on peut retourner les deux premières cartes, et obtenir `--`, puis retourner l'ensemble de la pile et obtenir `+++++`, c'est à dire une pile parfaitement ordonnée. Plus compliqué, si l'on prend la pile `+-++-` et que l'on retourne les quatre premières cartes, on obtiendra `--+--` (car on a non seulement changé l'orientation des cartes mais aussi l'ordre dans lequel elles étaient).

— **Entrée** : La première ligne de l'entrée donne le nombre de cas de test, `T`. Puis, les `T` cas suivent. Chaque cas de test est une suite de '`+`' et de '`-`' qui représentent la pile de cartes.

— **Sortie** : Pour chaque cas, la sortie sera une ligne contenant "`Case #x: `" (`x` commence à 1) suivi par le nombre d'opérations minimum à réaliser pour obtenir une pile triée correctement.

— **Limites** : Le nombre de cas de test (`T`) ne dépassera jamais 100. Et le nombre cartes (`S`) sur un cas de test ne dépassera jamais 100.

Fichier d'entrée

```
5
-
-+
+-
++++
--+-
```

Sortie sur `stdout`

```
Case #1: 1
Case #2: 1
Case #3: 2
Case #4: 0
Case #5: 3
```

Figure 1 – Exemple de fichier d'entrées/sorties.

Dans l'exemple donné, on voit que le premier cas nécessite seulement de retourner une fois l'ensemble de la pile. Le deuxième cas de retourner la première carte uniquement. Le troisième cas de retourner la première carte, puis l'ensemble de la pile. Le quatrième cas est déjà trié. Et, finalement, le cinquième cas peut se faire en retournant les deux premières cartes, puis les trois premières et enfin, toute la pile.

Enfin, vous trouverez des fichiers d'exemples plus aboutis à l'URL suivante :

`http://www.labri.fr/~fleury/courses/programming/master/exam/`

Questions

1. Commencez par écrire la partie du programme qui ouvre un fichier pour récupérer les données du problème. On suppose que l'interface utilisateur du programme se comportera comme suit :

   ```
   $> ./poquemon
   poquemon: error: No input file given !
   $> ./poquemon poquemon-xsmall.in
   Case #1:
   Case #2:
   ...
   ```

   En cas d'absence du fichier, le programme doit terminer en renvoyant une erreur sur `stderr` et avec un code de retour valant `EXIT_FAILURE`.

   Pour le reste du parsing, on supposera que nous avons toujours affaire à des fichiers "parfaits", sans erreur de syntaxe, ni oubli de la part de l'utilisateur. Inutile, donc, de passer du temps à essayer d'être robuste lorsque vous parsez le contenu du fichier. Par contre, on s'attend à ce que vous détectiez l'absence du fichier, ou un problème lors de l'ouverture (en lecture) de celui-ci.

   Il vous faut donc, dans l'ordre, vérifier la présence d'un argument sur la ligne de commande, ouvrir le fichier en lecture (et vérifier qu'il a bien été ouvert), récupérer le nombre de cas, puis faire en sorte de récupérer les éléments de chaque cas.

2. En fait, obtenir l'optimalité du nombre d'actions est assez simple car le nombre d'opérations est au moins le nombre d'alternations entre les séries de '`+`' et de '`-`' que l'on trouve dans la pile. Ainsi, il suffit de prendre le premier groupe de caractères semblables en haut de la pile, puis de les retourner, puis recommencer jusqu'à ce que les cartes soient toutes du même côté. Finalement, il faudra retourner l'ensemble de la pile si jamais on tombe sur une pile constituée uniquement de '`-`' ou laisser la pile telle qu'elle est si jamais on tombe uniquement sur des '`+`'.

   Par exemple, la pile `++---` a un alternation de '`+`' et de '`-`', il faudra donc une opération pour avoir les cartes toutes dans le même sens, puis une dernière opération pour retourner l'ensemble de la pile dans le bon sens. Un autre exemple, si l'on prend la pile `-+-+`, il y a trois alternations et il faudra trois opérations pour obtenir une pile ordonnée (on termine directement par une pile ordonnée, inutile de retourner l'ensemble de la pile une dernière fois).

   Programmez la résolution du problème en utilisant l'algorithme suggéré. La clarté du code, son efficacité ainsi que les commentaires que vous y mettrez seront aussi évalués.

2 Programmation Java (8 points)

Questions

1. Questions de cours :
   (i). Quel serait un équivalent de la notion d'héritage en C, et quelles différences avec la notion d'héritage offerte par Java ?
   (ii). Dans quelle situation est-il intéressant de déclarer qu'une méthode est abstraite ? (comme par exemple "`abstract void m();`").
   (iii). Donner un exemple de "*upcast*" et de "*downcast*" en langage C.
   (iv). Voici quelques déclarations en Java :

```
interface I { ... }
class C1 implements I { ... }
class C2 extends C1 {...}
class C1bis implements I { ... }
C2 x = new C2();
```

   Parmi les lignes suivantes, indiquer celles qui engendreront un problème et expliquer pourquoi :
   (a) `I a = x;`
   (b) `C1 b = (C1) a;`
   (c) `C2 c = (C1) a;`
   (d) `C2 d = (C2) a;`
   (e) `C1bis e = (C1bis) a;`
   (v). Quel est le principal effet bénéfique lorsqu'on utilise des structures de données *génériques*, i.e. avec des éléments typés par variables de types (comme par exemple dans `List<E>`), plutôt que des structures de données avec des éléments typés par `Object` (comme par exemple dans `List`).
2. Concevoir une architecture orientée-objet pour la simulation d'un orchestre musical. Les différents musiciens y seront munis d'instruments, capables de jouer des suites de sons. Il seront dirigés par un chef d'orchestre qui pourra leur indiquer quand commencer, s'arrêter, et jouer plus ou moins fort. Chaque musicien pourra être interrogé pour connaître quel son il produit à un temps donné.
   (i). Représenter votre conception au moyen d'un diagramme UML de classes simplifié.
   (*Contrainte : utiliser le logiciel* `dia`).
   (ii). Mettre en œuvre de manière minimale votre conception ci-dessus. La sortie de la simulation ne sera faite que d'un affichage de chaînes de caractères (`Do`, `Re`, `Mi`, `Fa`, `Sol`, `La`, `Si`) indiquant les sons joués par chaque musicien à chaque tic d'horloge, et cela pendant une durée fixée.