De: **François Pessaux** francois.pessaux@ensta-paris.fr
Objet: Re: [IN101] Éléments de correction.
Date: 20 février 2023 à 09:59
À: eleves1a_2022-23@ensta-paris.fr

Bataille

Q1.1
On veut décrire à haut niveau les grandes étapes du programme. Il n'est donc pas nécessaire de décrire dès maintenant le fonctionnement détaillé de chaque traitement. D'ailleurs comme, en accord avec les consignes, j'ai lu tout le sujet avant de me jeter à corps perdu dedans, je vois bien que les questions suivantes vont me demander de rentrer dans le détail de chaque sous-problème.
Donc, globalement je dois récupérer les cartes, les distribuer, jouer la partie puis afficher les résultats et finalement faire le ménage. Ah oui, mais pour récupérer les cartes, le nom du fichier m'est fourni en argument de ligne de commande. Donc il faut avant tout vérifier qu'il y en a. Bref, on résume :

```
programme :
  Vérifier qu'il y a des arguments et les récupérer.
  Charger les cartes, les distribuer et retourner les 2 paquets.
  Jouer la partie et récupérer le résultat final.
  Afficher le résultat.
  Libérer la mémoire.
```

Q1.2
Ah, déjà, au cas où je ne l'aurais pas senti à l'avance, je vois qu'il va falloir segmenter en faisant une fonction dont le seul travail est de préparer la partie. Chacun son boulot ! Elle a besoin du nom de fichier. Et elle va créer le ou les paquets de cartes. Oui, deux solutions : soit un paquet par joueur, comme « dans la vraie vie », soit un seul paquet dans lequel on piochera alternativement pour chaque joueur. Bon, on va se simplifier la vie en faisant 2 paquets comme cela on progressera de manière identique dans chaque. Sinon je sens bien qu'il va me falloir faire des `i` et `i + 1` un coup sur deux… ça va être plus glissant.
Donc, on retourne 2 tableaux.

Ah oui, en C les tableaux ne « contiennent » pas leur taille. Bon, il faut donc retourner leur taille aussi.
En résumé j'ai 3 trucs à retourner, ce que je ne peux pas faire directement en C. Je peux en « retourner par adresse » ou faire une structure. Faisons une structure, on aura moins à triturer du pointeur.

```
struct decks_t {
  int size ;     /* Taille d'un tas par joueur. */
  int *deck1 ;   /* Tas du joueur 1. */
  int *deck2 ;   /* Tas du joueur 2. */
};
```

Donc maintenant ma fonction prend une chaîne (le nom du fichier) et me retourne une telle structure.

Q1.3
La question est comment l'appelant de `load()` va pouvoir se rendre compte que le chargement a échoué ? Vu autrement, c'est comment `load()` va pouvoir signaler qu'elle a échoué ? Il faut trouver un moyen de retourner quelque chose dénotant une erreur. J'ai 3 champs dans ma structure. Je peux signaler un échec par une taille <= 0 ou au moins un des pointeurs « vide ». Le corollaire est que la fonction qui appelle `load()` devra vérifier, pour savoir si le chargement a réussi, si la taille est > 0 (si elle est égale à 0, ma foi, ça ne devrait pas poser de problème, on aura juste une partie vide) et si les deux pointeurs sont bien différents de NULL.

Q1.4
Qu'est-ce qui peut mal se passer quand je récupère les cartes et j'en fais mes paquets ? Qu'est-ce qui est impliqué dans ce travail ?
Il y a la manipulation de fichier et le stockage en mémoire des données.
Pour le fichier, il faut l'ouvrir et lire dedans. Ça peut mal se passer dans les deux traitements.
Pour le stockage en mémoire, il faut que j'ai de la mémoire. Et si je n'arrive pas à en obtenir ça va mal se passer. Eh oui, vu que je ne connais pas à l'avance le nombre de cartes (au moment où j'écris mon programme), je ne peux pas me faire un tableau de taille fixe dans mon programme. Il va falloir que j'alloue dynamiquement. `malloc()` arrive à grands pas… et `free()` aussi à un moment (je garde ça dans la tête).

On récapitule :
- Impossible d'ouvrir le fichier (plein de bonnes raisons : droits d'accès, fichier inexistant…)
- Impossible d'obtenir de la mémoire
- Problèmes lors de la lecture. Oui mais lesquels ?
    - Si je lis une taille négative, je ne vais pas faire un tableau de taille négative !
    - Et si je lis une taille positive, vu que je devrai couper la suite des cartes en deux, il faut qu'elle soit paire sinon je ne pourrai pas.

- Après, quand je lis mes cartes, si je n'arrive pas à en lire assez par rapport au nombre initialement promis, ça va coincer.

Q1.5
On a dit qu'on récupérait les infos depuis un fichier. Il va donc falloir l'ouvrir. Ensuite, vu la forme du fichier, je dois y trouver un nombre de carte puis les cartes. Il va donc falloir que je récupère et stocke ces infos. Et quand je lis, idéalement je dois vérifier la validité de ce que j'ai lu. À la fin il faudra que je retourne ma structure représentant la partie « prête à l'emploi ». Ah, et puisque j'ai terminé avec le fichier, je dois le fermer.
Faisons ça dans l'ordre…

```
load (filename) =
  Ouvrir fichier filename en lecture.
  Si erreur d'ouverture, retourner erreur.
  Lire nombre de cartes à lire -> nb_cards.
  Si nb_cards <= 0 ou impair, retourner erreur.
  Allouer tableaux t1 et t2 de nb_cards / 2 entiers.
  Si l'un au moins est nul,
    Libérer l'éventuel non nul.
    Fermer fichier.
    Retourner erreur.
  Boucler de i=0 à nb_cards / 2 exclus,
    t1[i] <- lire entier dans fichier.
    Si erreur lecture, libérer tableaux, retourner erreur.
    t2[i] <- lire entier dans fichier.
    Si erreur lecture, libérer tableaux, retourner erreur.
  Fermer fichier.
  Retourner t1, t1 et nb_cards / 2.
```

Si j'étais vraiment pointilleux, je pourrai même vérifier s'il n'y a pas des cartes en trop. Bon, on verra si j'ai le temps à la fin.

Q1.6
Cette fonction doit jouer la partie. Il faut donc qu'elle prenne une partie en entrée. Ca tombe bien, `load()` en fournit justement une. En sortie, l'énoncé me dit texto « *et retourner l'état final de celle-ci (i.e. les informations demandées dans la sortie finale, qui seront affichées par la fonction principale -- votre main)* ».
Donc ma fonction n'affiche pas, elle doit bien retourner quelque chose. Mais quoi ?
In fine j'aurai besoin d'afficher qui est le vainqueur et les scores. Je pourrais donc retourner ces 3 informations (il y a 2 scores). Mais en fait, est-ce que j'ai besoin de retourner tout ça ? Avec juste les 2 scores je pourrai ensuite simplement déterminer le vainqueur. Autant se simplifier la vie et ne retourner que les 2 scores.
Rebelotte, en C on ne peut retourner qu'une seule valeur. Comme précédemment, on a le choix entre du passage par adresse ou se faire une structure. On ne change pas une équipe qui gagne, on se refait une structure pour se simplifier la vie.

```
struct scores_t {
  int s1 ;
  int s2 ;
};
```

Donc ma fonction prend une `struct desks_t` et retourne une `struct scores_t`.
Si je voulais retourner les scores par adresse, j'aurais eu un type void `scores_t play (int *deck1, int *deck2, int size, int *score1, int *score2)` à la place.

Q1.7
Je remarque que par défaut, à chaque tour le gagnant augmente son score de 1 par carte qu'il a posée. Par contre, en cas de bataille, il faut aller chercher d'autres cartes sans savoir encore qui va gagner les points des cartes déjà posées.
Il va donc me falloir une variable permettant de mémoriser le gain en suspens en cours de bataille. Et lors d'une bataille, à chaque bataille deux points supplémentaires sont mis en jeu.
Donc je n'ai qu'à avancer dans chaque paquet en même temps, mettre à jour les score lorsqu'il n' a pas ou plus de bataille et sinon, en cas de bataille, accumuler les points en suspens. Comme ça, je n'ai pas besoin d'aller voir dans « le futur » ce sera le tour prochain de boucle qui me dira quoi faire des scores et des points en suspens. Ou un tour encore plus lointain si l'on enchaîne les batailles.
Pas besoin non plus de « boucler » tant qu'il y a bataille puisque je mets les points en suspens : on avance dans la tableau de cartes même s'il y a bataille. Le seul truc est de penser à avancer d'une carte en plus en cas d'étape de bataille pour sauter la carte posée face cachée.

```
play (deck1, deck2, size) =
  score1 <- 0
  score2 <- 0
  pending_gain <- 0
```

```
Pour i de 0 jusqu'a size exclu
  c1 <- carte i du joueur 1
  c2 <- carte i du joueur 2
  Si c1 < c2 alors
    score1 <- score1 + (pending_gain + 1)
    pending_gain réinitialisé à 0
  Sinon si c2 > c1 alors
    Même chose que le cas précédent mais avec score2
  Sinon
    pending_gain <- pending_gain + 2
    i <- i + 1 pour sauter une carte
Retourner les 2 scores.
```

Q1.8
On a vu qu'un des deux scores seulement augmente à chaque étape gagnée. Et il augmente du nombre de cartes posées par un joueur. Chaque joueur a « nombre de cartes initial » / 2. Donc la somme des deux scores devrait être égale à « nombre de cartes initial » / 2. Ah, mais attention, si la partie se termine par une bataille inachevée par manque de cartes… les points en suspens ne sont pas attribués. C'est donc la seule exception.

Q1.9
Là c'est du code…

Petite remarque que j'ai notée dans plusieurs rendus : `fscanf()` ne retourne pas forcément -1 en cas d'erreur. Nous avons vu en TD qu'il retourne le nombre de « %truc » effectivement lus. Il retourne -1 uniquement si la lecture a échoué pour d'autres raisons qu'une erreur de conversion en « truc » (par exemple fin de fichier ou fichier devenu inaccessible pour quelconque raison).
Donc si l'on veut vérifier le succès de la lecture de 2 entiers, il faut vérifier :

```
res = fscanf (in, "%d %d" , &x, &y) ;
if (res != 2) erreur ;
```

Robot

L'énoncé me dit clairement que mon robot ne connaît pas les chaînes, que le « programme » est reçu sous forme d'un entier, que le robot doit « *retourner une double indication disant si `a l'issue de sa promenade…* ». Donc je vais devoir faire une fonction qui prend un entier et retourne le statut de la mission.

C'est quoi ce statut ? C'est « de retour » ou « perdu » et le nombre de cases qu'il a traversées. J'ai donc deux infos à retourner.
En première approche, « de retour » ou « perdu » je peux l'encoder par un booléen. Gardons ça en mémoire.
Bref, on va se refaire une structure pour retourner ces infos.

Réfléchissons un peu au traitement… Il faut que je décortique mon nombre pour en extraire chacun des chiffres. Ai-je besoin de les avoir tous en mémoire ? Ben non : je peux les prendre au fur et à mesure et regarder à chaque fois quel mouvement faire.

Maintenant, dans quel sens dois-je les récupérer ces nombres ? De droite à gauche ou l'inverse ? En fait, ça n'a aucune importance : que je fasse le chemin dans un sens ou dans l'autre, si je dois revenir à la position initial j'y reviendrai. Donc je vais choisir le sens de décomposition qui me simplifiera la vie.
Pour récupérer le chiffre de droite, il me suffit de prendre le reste de la division par 10 (le modulo). Et pour passer au chiffre suivant, je divise par 10. Et je recommence tant que je ne suis pas à 0. C'est simple et ça égrène les chiffres de droite à gauche. Ah tiens, dans l'ordre inverse du « programme ». Mais on a vu que ce n'était pas gênant, tant mieux !
Si l'on avait absolument voulu égrener les chiffre de gauche à droite, il aurait fallu procéder différemment car il aurait fallu diviser par 10 en chaîne jusqu'à « buter » sur le chiffre le plus à gauche et récupérer chaque autre chiffre en redescendant de cette suite de divisions. Clairement, un algorithme récursif aurait été nécessaire.

Maintenant que je sais récupérer mes chiffres un par un, que dois-je en faire à chaque itération ? Ben « faire bouger » le robot, donc le déplacer en *x* / *y*. Je n'ai qu'à maintenir 2 variables x et y qui représentent la position de mon robot. Initialement elles sont à 0. Et si je suis revenu à mon point de départ à la fin du « programme », c'est qu'elles vaudront 0.

Comment on se déplace ? Ben si le chiffre est 0, je ne bouge pas, si c'est 1 je fais `x <- x + 1` et `y <- y + 1`, etc.
Et si le chiffre est 9 ? Oups, c'est un cas d'erreur ! Il va me falloir le signaler dans le retour de la fonction. Ah, donc son retour ne peut pas être un simple booléen car il me faut signaler au choix : « de retour », « perdu » ou « erreur ». En C, je peux définir un `enum`. Ou sinon je peux manuellement encoder ces valeurs sur un entier.

Il faut aussi que je retourne le nombre de cases traversées. Donc à chaque chiffre je dois incrémenter un compteur. Chaque ? Ah ben non, en cas de 0, ce n'est pas une erreur mais on ne bouge pas. Donc on ignore les 0. Ainsi, la longueur du chemin n'est pas la longueur du nombre en base 10 : c'est elle mais en ne comptant pas les 0.

Globalement j'arrive donc à :

```
robot (prog) =
  Initialiser tableaux dx et dy.
  x, y, nb_cells <- 0.
  Tant que prog <> 0
    c <- prog % 10.
    Si c > 8 alors retourner erreur
    Mettre à jour x et y en fonction de c
    Si c <> 0 alors nb_cells <- nb_cells + 1
    prog <- prog / 10
  Si x et y nuls, retourner (IsBack, nb_cells)
  Sinon retourner (IsLost, nb_cells)
```

Il me reste à écrire la partie `Mettre à jour x et y en fonction de c`.
Je peux faire une cascade de 8 `if-else`. C'est long à écrire, c'est répétitif, bof :/
En réfléchissant, je peux réduire en factorisant par des « ou » et des « et » les cas où x augmente, où x diminue, et pareil pour y. Je n'ai plus que 4 `if` 2 `else`. C'est déjà plus lisible.
Ah mais sinon, je sais que mes chiffres légaux sont entre 0 et 8. Je peux donc m'en servir comme indices dans 2 tableaux donnant le déplacement en *x* et en *y* pour chaque chiffre. Comme cela, plus de tests :)

Je regarde sur un morceau de papier « où je vais pour chaque chiffre » et j'en déduis :

```
  int dx[9] = { 0, 0, 1, 1, 1, 0, -1, -1, -1 } ;
  int dy[9] = { 0, 1, 1, 0, -1, -1, -1, 0, 1 } ;
```

Et donc ma mise à jour de x et y devient simplement :

```
    x += dx[digit] ;
    y += dy[digit] ;
```

J'arrive au bout. Il me reste à écrire la fonction principale. Elle récupère l'argument sur la ligne de commande. Elle doit donc commencer par vérifier qu'il y en a bien un. Puis le transformer en entier long comme décrit dans l'énoncé. Puis appeler la fonction `robot()` et afficher le statut en fonction des 2 informations reçues en retour.

```
main () =
  Vérifier qu'il y a des arguments et les récupérer.
  Convertir l'argument en entier.
  Appeler robot () et récupérer (statut, nb_cells).
  Afficher message en fonction de si perdu, de retour ou erreur.
```