Pablo Rauzy
pr@up8.edu
https://pablo.rauzy.name/teaching/mdlp

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

Méthodologie de la programmation

Université Paris 8 – Vincennes à Saint-Denis
UFR MITSIC / L1 informatique

Séance h (TP) : Jeu de pendu en C

N'oubliez pas :
- Les TPs doivent être rendus par courriel au plus tard la veille de la séance suivante avec "[mdlp]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[mdlp] TPh Rauzy".
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu dans une archive, par exemple un tar gzippé obtenu avec la commande `tar czvf NOM.tgz NOM`, où `NOM` est le nom du répertoire dans lequel il y a votre code (idéalement, votre nom de famille et le numéro de la séance, par exemple "rauzy-tph").
- Si l'archive est lourde (> 1 Mo), merci d'utiliser `https://bigfiles.univ-paris8.fr/`.
- Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l'archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N'hésitez jamais à chercher de la documentation par vous-même sur le net!**

Dans ce TP :
- Programmation modulaire en C.
- Compilation avec Makefile.

Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/mdlp/sh-tp/`.
2. Récupérez les fichiers nécessaires depuis la page web du cours, ou directement en ligne de commande avec `wget https://pablo.rauzy.name/teaching/mdlp/seance-h_tp.tgz`.
3. Une fois que vous avez extrait le dossier de l'archive (par exemple avec la commande `tar xzf seance-h_tp.tgz`), renommez le répertoire en votre nom (avec la commande `mv mdlp_seance-h_tp` *`votre-nom`*). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

État des lieux.

1. Dans ce TP le but est de faire fonctionner un jeu de pendu écrit en C, puis d'automatiser correctement sa compilation.

   Le but du jeu du pendu est de deviner un mot. Au départ, seul le nombre de lettre du mot est connu. À chaque tour, la joueuse propose une lettre. Si cette lettre est dans le mot, alors on lui indique à quelle(s) position(s). Sinon, il est un peu plus proche d'être pendu. la joueuse à le droit de se tromper 10 fois avant d'être pendu (cela correspond aux 10 traits nécessaires à dessiner un petit bonhomme pendu). la joueuse gagne si il trouve le mot avant d'être pendu, et perd sinon.

   D'ores et déjà, on peut commencer à réfléchir à ce qui sera nécessaire pour ce projet :
   - un moyen de sélectionner un mot au hasard dans une liste,
   - manipuler une chaîne de caractères (pour cacher les lettres du mot puis les afficher au fur et à mesure),
   - interagir avec la joueuse (lui afficher des messages, lire ses entrées au clavier),
   - gérer l'état du jeu (quelles lettres ont déjà été essayée, combien il reste de coup au joueur, ...).

   On va donc avoir besoin de 3 modules :
   - la gestion les mots (ce qui correspond aux deux premiers éléments de la liste ci-dessus),
   - l'interaction avec la joueuse,
   - la gestion du jeu.

   Instinctivement, on pourrait avoir envie de ne faire qu'un seul module pour les deux derniers. Cependant, c'est toujours une bonne idée de séparer la *logique* (ici, l'implémentation des règles du jeu et la gestion de son état) de l'*interface* (ici, les interactions avec la joueuse). D'une part, cela permet d'obtenir un code plus facilement maintenable, par exemple on aura pas des `printf` en pagaille au milieu du code du jeu proprement dit. D'autre part, cela permet de faire évoluer le projet plus efficacement, par exemple si on veut finalement que le jeu soit

jouable en plusieurs langues ou si on veut plus tard interagir avec la joueuse dans une interface graphique plutôt que dans un terminal, on aura juste à changer le module d'interaction avec l'utilisateur et garder la logique du jeu intact.

On retrouve donc dans le code trois modules `word`, `pendu`, et `display`. On a aussi décidé (un peu artificiellement), de rajouter un module `charlist` pour gérer des listes de caractères qui seront utilisées pour gérer l'état du jeu. Il y a aussi un fichier `main.c` qui contient la fonction `main`.

Le premier but de ce TP va être pour vous d'écrire l'implémentation du module `charlist`.

→ Ouvrez et lisez le code des différents modules. Identifiez les endroits où le module `charlist` est utilisé et essayez de comprendre comment.

2. Le jeu ne fonctionne évidemment pas encore, mais on peut déjà le compiler. Le fait que les fonctions du module `charlist` ne soient pas encore implémentées va provoquer beaucoup de "warning" mais pas d'erreurs (le langage C est très permissif, pour votre mal), donc la compilation continue jusqu'au bout.

   → Compilez le projet avec la commande `gcc -Wall *.c -o pendu`.

   L'option `-Wall` (pour "warning all") active l'affichage de plus de warnings que par défaut, c'est toujours une bonne idée car les warnings ne sont pas présent pour rien.

3. → Exécutez le jeu que vous venez de compiler. Que se passe-t-il ? Comment expliquez vous cela ?

4. → Dans les fichiers entêtes (`.h`) qu'est ce que la garde d'inclusion ? À quoi sert-elle ?

Exercice 2.

Implémentation du module `charlist`.

1. Une liste chaînée est
   — soit une liste vide,
   — soit une paire composée d'un élément de la liste (pour nous, un `char`), et de la suite de la liste, c'est à dire un pointeur vers une autre liste chaînée (potentiellement vide si on est le dernier maillon de la liste chaînée).

   → Déclarez les deux champs `head` et `tail` de la structure `CharList` en leur donnant les bons types.

2. En C, on va représenter la liste vide par le pointeur `NULL`. La fonction `CharList_nil` renvoie une liste vide.

   → Implémentez la fonction `CharList_nil`.

3. La fonction `CharList_cons` prend en argument un caractère et une liste, et créée une nouvelle liste commençant par ce caractère et suivi de la liste reçu en argument.

   Pour cela, la fonction doit allouer de la mémoire pour une structure `CharList` pour contenir le nouveau maillon de tête de la liste chaînée, en utilisant la fonction `malloc` comme on l'a vu en cours.

   → Implémentez la fonction `CharList_cons`.

4. La fonction `CharList_head` renvoie le caractère contenu dans le premier maillon de la liste chaînée reçue en argument.

   → Implémentez la fonction `CharList_head`.

5. La fonction `CharList_tail` renvoie la suite de la liste chaînée reçue en argument.

   → Implémentez la fonction `CharList_tail`.

6. La fonction `CharList_delete` libère la mémoire utilisée par une liste chaînée, c'est à dire de chacun de ses maillons, en utilisant la fonction `free`.

   → Implémentez la fonction `CharList_delete`.

7. → Que fait la fonction suivante :

```c
void
CharList_print (CharList *self)
{
  if (self == NULL) {
    printf("\n");
    return;
  }
  printf("%c", self->head);
  CharList_print(self->tail);
}
```

8. Il faut régulièrement tester votre code !

   → Créez un fichier `test.c` avec une autre fonction `main` dans laquelle vous ne lancez pas le jeu de pendu mais où vous testez que les fonctions que vous avez écrites jusqu'à présent fonctionnent bien.

Exercice 3.

Utilisation des listes chaînées pour gérer des ensembles.

1. En fait, dans notre jeu de pendu, ce dont on a vraiment besoin sont des ensembles : lorsque la joueuse propose une lettre, on veut simplement vérifier si il/elle ne l'a pas déjà proposée, et sinon si la lettre fait partie de celles qui restent à deviner.

   On va pour cela utiliser nos listes chaînées (ce qui n'est pas optimal en pratique, c'est vraiment juste un prétexte pour l'exercice du TP), mais on va du coup devoir écrire des fonctions de manipulation de ces listes qui nous permette de les utiliser comme des ensembles.

   Un ensemble peut contenir plusieurs éléments, mais chacun doit être unique. On peut insérer un élément dans un ensemble, enlever un élément d'un ensemble, et vérifier l'appartenance d'un élément à un ensemble.

2. La fonction `CharList_has` reçoit une liste et un caractère en arguments, et renvoie vrai si la liste contient le caractère, faux sinon.

   → Implémentez la fonction `CharList_has`.

3. La fonction `CharList_insert` reçoit une liste et un caractère en arguments, et renvoie une nouvelle liste qui contient le caractère si celui-ci n'était pas déjà dans la liste reçue en argument, sinon renvoie simplement cette dernière.

   → Implémentez la fonction `CharList_insert`.

4. La fonction `CharList_remove` reçoit une liste et un caractère en arguments, et renvoie une nouvelle liste qui contient les mêmes éléments que celle reçue en argument sauf qu'elle ne contient plus le caractère.

   Attention, n'oubliez pas de libérer la mémoire des maillons supprimés de la chaîne.

   → Implémentez la fonction `CharList_remove`.

5. Maintenant le jeu de pendu peut fonctionner correctement.

   → Compilez le jeu et testez le.

Exercice 4.

Makefile.

1. → Écrivez un Makefile le plus générique possible pour compiler et nettoyer ce projet.
2. → Ajoutez une cible à votre Makefile pour compiler un exécutable `testcharlist` qui utilise le `main` de votre fichier `test.c`.

   Si éventuellement vous avez besoin d'autres cibles intermédiaires, n'oubliez pas de les ajouter aussi.

Exercice 5.

Rajoutez des mots au jeu.

1. → Décrivez ce qui se passe dans la fonction `Word_new`. N'hésitez pas à faire appel aux pages de manuel de la librairie standard de C (`man` *`fonction`*).
2. → Ajoutez des mots dans le fichier `dict.txt` de tel sorte à ce que le jeu puisse les choisir.