Séance 5 et 6 : Unix, compléments sur les pointeurs

L2 Informatique – Université Côte d'Azur

Exercice 1 — Numérotons les lignes

Le but de cet exercice est de réimplémenter la commande `nl` d'Unix. Cette commande, comme son nom l'indique, **N**umérote les **L**ignes d'un fichier. Concrètement, il y a deux façons de l'utiliser. Premièrement via l'entrée standard ; par exemple, le commande « `echo -e "ligne1\nligne2\nligne3" | ./nl` » devra afficher :

stdin/stdout
```
1: ligne1
2: ligne2
3: ligne3
```

Deuxièmement en indiquant en paramètre le nom du fichier : « `./nl Makefile` » affichera ainsi :

stdin/stdout
```
1: CC = gcc
2: OPTIONS = -Wall -pedantic -ansi
3:
4: all: nl
5:
6: %:%.c
7:     $(CC) $(OPTIONS) $< -o $@
```

Si plusieurs fichiers sont donnés en paramètre, il faudra les afficher les uns à la suite des autres (en continuant à incrémenter les numéros de lignes). Si un des fichiers en paramètre ne peut pas être ouvert, il faudra afficher une erreur avant de quitter proprement le programme.

Exercice 2 — Liste chaînées et récursion

Le but de cet exercice est de programmer les fonctions de base permettant de définir et de manipuler des listes chaînées. Dans cet exercice, on implémentera les listes chaînées pour les utiliser dans des algorithmes récursifs comme on peut en rencontrer dans des langagues comme OCaml.

Partie I : création des types

1. Commencez par créer une structure `cellule` contenant deux champs : un champs « `contenu` » de type entier et un champs « `suivant` » représentant un pointeur vers une autre cellule.
2. Pour simplifier, les types des fonctions, redéfinissez avec `typedef` le type « pointeur vers `cellule` » sous le nom `liste_chainee`.

Partie II : fonctions d'accès au type

Écrivez les six fonctions suivantes permettant de manipuler notre nouveau type.

3. `liste_chainee nouvelle_lc()` renvoyant une liste chaînée vide.
4. `int est_vide(liste_chainee L)` indiquant si la liste est vide.
5. `int head(liste_chainee L)` renvoyant la valeur du premier élément de la liste (ne fonctionne que si `L` est non vide).

6. `liste_chainee tail(liste_chainee L)` renvoyant la suite de la liste (ne fonctionne que si `L` est non vide).
7. `liste_chainee ajout_lc( int x, liste_chainee L)` qui renvoie une nouvelle liste dont le premier élement est `x` et dont la suite est `L`.
8. `void liberer_lc(liste_chainee L)` libérant la mémoire associée à la liste.

Partie III : utilisation de notre type

Dorénavant, on oublie comment sont codées les listes chaînées et **on a uniquement le droit d'utiliser les fonctions de la partie II**. Tous les codes devront être récursifs :

- cas de base : la liste est vide
- cas récursif : la liste se décompose sous la forme `head(L)` et `tail(L)`.

On pourra créer une liste de la façon suivante.

```
liste_chainee L = nouvelle_lc();
L = ajout_lc(20,L); /* head(L) = 20 et tail(L) = NULL          */
L = ajout_lc(1,L);  /* head(L) = 1  et tail(L) = [20]→NULL    */
L = ajout_lc(17,L); /* head(L) = 17 et tail(L) = [1]→[20]→NULL */
```

Écrire et tester les fonctions suivantes :

9. la fonction `affiche_lc` qui affiche une liste chaînée. Une fois cette fonction écrite, tester abondamment les fonctions précédentes. La liste de l'exemple précédente pourra être affichée sous la forme :
`[17] → [1] → [20] → NULL`.
10. la fonction `longueur_lc` qui renvoie le nombre d'élements de la liste donnée en paramètre.
11. la fonction `copie_lc` qui renvoie une copie de la liste donnée en paramètre.
12. la fonction `ajout_fin_lc` renvoie une copie de la liste auquel a été ajouter un élement à la fin.
13. la fonction `max_lc` qui affiche le plus grand élément de la liste. Si la liste est vide on renverra pas convention `-INT_MAX-1` qui est le plus petit entier possible (`INT_MAX` est définie dans la bibliothèque `limits.h`).
14. la fonction `map_lc` qui prend en argument une liste et un pointeur vers une fonction et renvoie une nouvelle liste obtenue en appliquant la fonction à tous les élements de l'ancienne liste. Pour une liste `L` de la forme :

    `[a] → [b] → [c] → [d] → NULL`

    l'appel de `map_lc(L, f)` renvoie une nouvelle liste valant :

    `[f(a)] → [f(b)] → [f(c)] → [f(d)] → NULL`

Exercice 3 — Liste chaînée et programmation impérative

Dans cet exercice, on conserve le type `cellule` de l'exercice précédent, mais cette fois ci, une liste sera une structure contenant un unique champs « `premier` » pointant vers la première cellule de la liste.

La différence avec l'exercice précédent est qu'ici, l'objet `L` ne changera pas (même si la mémoire vers laquelle il pointe évoluera au fils du temps). Si les deux structures sont proches, l'usage va être assez différent car dans cet exercice la **récursion sera interdite**.

On pourra créer une liste de la façon suivante.

```
liste L = nouvelle_liste();
ajout_debut(&L,20);
ajout_debut(&L,1);
ajout_debut(&L,17);
```

Partie I : création des types

1. Créez ainsi un type `liste` contenant un champs de type `cellule*`
2. Écrire les 5 fonctions suivantes et testez les proprement.
   (a) `liste nouvelle_liste()`
   (b) `void liberer_liste(liste L)`
   (c) `int est_vide(liste L)`

(d) `void ajout_debut(liste *L, int x)`

(e) `void afficher_liste(liste L)`

Partie II : autres fonctions de base

Écrire et testez les trois fonctions suivantes. On fera bien attention à la gestion mémoire.

3. `void ajout_fin(liste *L, int x)`
4. `int longueur(liste L)`
5. `int pop_liste(liste *L, int *x)` : supprime le premier élément de la liste et le stocke dans le pointeur `x`. Renvoie `-1` en cas d'erreur.

Partie III : fonctions plus avancées

6. `liste tableau_vers_liste(int tableau[], int taille)` construit une liste à partir d'un tableau.
7. `int lire_valeur(liste L, int i, int *x)` lit la valeur d'indice `i` de la liste et la stocke dans le pointeur `x`. Renvoie `-1` en cas d'erreur.
8. `int inserer(liste *L, int x, int i)`. Insère une valeur à l'indice `i` demandé. Renvoie `-1` si l'indice n'est pas valide.
9. `int max(liste L, int *x)` stocke dans `x` la plus grande valeur de la liste `L`. Si la liste est vide renvoie `-1`. Sinon, renvoie l'indice de cette valeur.
10. `void map(liste L, int f(int))` applique la fonction `f` à chaque élément de `L`.
11. `void supprimer(liste * L, int f(int))` supprime de la liste `L` toutes les valeurs `x` telles que `f(x)` soit vrai.

Exercice 4 — Recodons les utilitaires Unix

Écrivez un programme `mywc` (qui est une version semblable au `wc` du Shell), qui compte les lignes, les mots et les caractères d'un fichier, et écrit les résultats sur la sortie standard. Un mot est une suite de caractères quelconques, encadré par un ou plusieurs espaces.

On fera attention d'avoir un comportement similaire au véritable `wc`. Les fichiers pourront être lus soit depuis l'entrée standard soit donnés en paramètre. Dans les deux cas l'affichage doit être le même que celui la commande d'origine. On implémentera aussi les trois options :

1. `-c` (ou `--bytes`),
2. `-l` (ou `--lines`),
3. `-w` (ou `--words`),