UNIVERSITÉ D'ORLÉANS **L3 Info**

Informatique Année **2016 - 2017**

Série de Travaux Dirigés : **1** *- Les chaînes de caractères - TD NOTÉ*

L'objectif de ce TD est de manipuler les chaînes de caractères de taille variable à partir d'une structure permettant de ne pas avoir à gérer ni un caractère de fin de chaîne ni les allocations et les dé-allocations. Par la suite un caractère désignera un caractére alphabétique en minuscule ou majuscule sans les caractères spéciaux et les caractères accentués (donc les char du C).

Le TD est divisé en deux parties. La première consiste à mettre en place la structure `chaine`, à implémenter les fonctions nécessaires à leurs manipulations et à vérifier qu'elles fonctionnent. La deuxième partie a pour objectif de développer le jeu Motus à partir d'une nouvelle structure s'appuyant sur la structure `chaine`.

Exercice 1. La structure chaine

À partir de l'archive `TD1.tgz` disponible sur Celene, vous disposez des fichiers suivants :

— `chaine.h` contenant les entêtes des fonctions à développer et le typedef de la structure `chaine`.
— `chaine.c` contenant les include nécessaires, la structure `chaine` et un squelette des fonctions à développer.
— `test_chaines.c` et `test_chaines_in.txt` pour tester vos fonctions.
— `Makefile` pour compiler et exécuter les différents tests liés à la notation.

Les fichiers contiennent les commentaires nécessaires aux développements demandés. La structure `chaine` est définie sur la Fig. 1

```
struct chaine {
  unsigned int taille;
  char* tab;
};
```

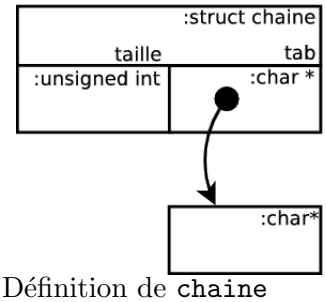

FIGURE 1 – Définition de `chaine`

Ainsi, une `chaine` est définie par une taille et un tableau de caractères de longueur exactement cette taille c'est-à-dire la taille minimale pour la chaîne correspondante. De plus il est demandé de travailler à partir du

`typedef struct chaine* chaine` .

Dans `chaine.h` l'ordre de déclarations des fonctions est l'ordre proposé pour le développement. Parmi les fonctions demandées voici quelques précisions

1. Les fonctions `void chaine_afficher(FILE* f,chaine ch)` et `chaine chaine_lire(FILE* f, unsigned int taille)` prennent en paramètre un stream et doivent permettre la lecture ou l'écriture sur la sortie et l'entrée standards ainsi que sur un fichier.
2. `void chaine_concatener(chaine ch1, chaine ch2)` permet la concaténation où ch1 vaut ch1.ch2 à la sortie de la fonction.
3. `bool chaine_appartenir(char c, chaine ch, int* i)` renvoie un booléen indiquant l'appartenance du caractère `c` dans `ch` et l'entier `i` contient la place du caractère dans la chaîne.

Enfin pour les tests, le Makefile doit être utilisé de la manière suivante

— `make test_chaines` permet de compiler et d'engendrer l'exécutable du programme `test_chaines.c` fourni.
— `make test` permet d'exécuter le test et de comparer avec le résultat à obtenir.
— `make memoire_chaines` exécute le test avec valgrind et vérifie si la mémoire est correctement gérée.

Le fichier `chaines.h` ne doit pas être modifié. Dans le compte rendu vous devrez bien indiquer les fonctions qui ne fonctionnent pas (si c'est le cas) et le bilan du test avec valgrind.

Exercice 2. Le jeu Motus

Le jeu Motus est similaire au Master mind^{TM} mais avec des mots à la place de suites de couleurs. Ainsi le joueur propose un mot de 6 lettres (par exemple) pour lequel on indique les lettres à la bonne place et les lettres appartenant au mot. Le joueur peut ainsi tenter de trouver le bon mot en s'aidant de ces indications.

Dans cette deuxième partie, vous devez développer ce jeu à partir d'une structure déjà définie. Vous disposerez également d'une fonction d'affichage du jeu et il vous reste à définir 4 fonctions pour créer, détruire le jeu et pour réaliser son déroulement.

Vous trouverez toujours dans l'archive `TD1.tgz` les fichiers suivants

— `motus.h` avec les entêtes

— `motus.c` avec la définition de la structure et l'implémentation de la fonction d'affichage.

— `jeu.c` avec le déroulement du jeu et qui contient également la génération du mot à trouver par lecture dans le fichier test_jeu_in.txt.

La structure est la suivante

```
struct motus {
  unsigned int t_mot;
  unsigned int nb_essai;
  chaine mot;
  chaine propositions;
  chaine resultats;
};
```

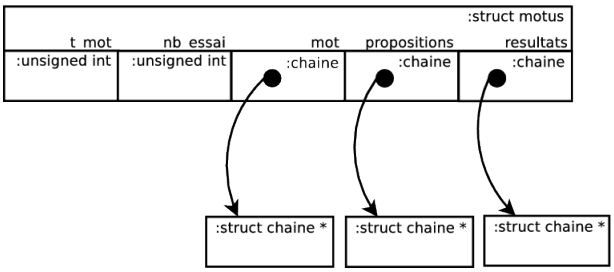

FIGURE 2 – La structure `motus`

Ainsi le joueur devra trouver un mot de longueur `t_mot` en au plus `nb_essai`. Le mot à trouver est stocké dans `mot` et les propositions sont concaténées dans `propositions`. La `chaine resultats` permet d'adopter un code utile par exemple pour l'affichage pour différencier les lettres qui sont à la bonne place des lettres qui sont présentes mais mal placées ou des lettres qui n'appartiennent pas au mot. La `chaine resultats` est aussi la concaténation pour chaque proposition du code correspondant. La fonction d'affichage utilise le code suivant :

— `=` lorsque la lettre est à la bonne place,

— `!` lorsque la lettre appartient au mot,

— `.` lorsque la lettre n'est pas dans le mot recherché.

Voici quelques exemples :

mot à trouver	`citron`	`livres`	`absolu`
mot proposé	`castor`	`lapins`	`risque`
code	`=..!=!`	`=..!.=`	`..=.!.`

Vous pourrez implémenter une première version du jeu qui suppose que les mots utilisés sont sans lettre en double et que le joueur propose des mots également sans lettre en double (exemples ci-dessus). Ensuite, vous pourrez faire évoluer le jeu vers sa version complète en tenant compte des lettres multiples dans le mot à chercher et les propositions. Voici quelques exemples avec des lettres en double lorsque la priorité est donnée aux lettres bien placées :

mot à trouver	`filles`	`petite`	`absolu`
mot proposé	`filial`	`pattes`	`passif`
code	`===..!`	`=.=!!.`	`.!=...`

On vous propose de réaliser le jeu à l'aide de deux fonctions :

1. `chaine chaine_code(chaine ch, chaine mot, bool* gagne)` qui permet de définir une `chaine` représentant le code proposé pour tester l'appartenance des lettres du mot `ch` dans le mot de référence `mot`. Le booléen `gagne` permet également de savoir si le mot proposé correspond au mot recherché.
2. `void motus_jeu(motus m)` qui permet de dérouler le jeu en fonction du nombre d'essais possibles.

Enfin pour les tests, le Makefile doit être utilisé de la manière suivante :
— `make jeu` permet de compiler et d'engendrer l'exécutable à partir du programme `jeu.c` à compléter,
— `make test_jeu` permet de lancer le jeu avec des mots de taille 6, 6 essais et la lecture du mot à trouver dans un petit fichier texte `test_jeu_in.txt` ,
— `make memoire_jeu` exécute le test avec valgrind et vérifie si la mémoire est correctement gérée.

Les fichiers `motus.h` et `jeu.c` ne doivent pas être modifiés.

Compte-rendu

Dans un fichier `cr.pdf` :
— donner la composition du binôme,
— expliquer et justifier la complexité de la méthode que vous utilisez pour engendrer `resultats`,
— écrire l'état d'achèvement du TP, si le jeu fonctionne, comment se passent les tests... et
— un bilan plus personnel.

Rendu

La commande

`make rendu`

engendre un fichier archive `rendu.tgz` qui contient les `*.c`, `*.h`, `cr.pdf` et `[Mm]akefile` de votre répertoire.

Ce fichier qui doit être remonté sous Celene, au plus tard 15 minutes après la fin du TP.

Aucun autre format ni nom de fichier n'est accepté pour la remise.