Projet — Le boss de fin

Le but de ce projet est de regrouper **tous** les éléments vus lors des CMs, TDs et TPs précédents. **Le projet doit être réalisé en binôme** : un dépôt `git` sera créé pour chaque binôme, et le rendu se fera **prioritairement** via `git`. Le code doit être commenté et son exécution avec `valgrind` **doit contenir un minimum d'erreurs** (idéalement aucune, évidemment)). **Le dernier exercice du TP6 doit être fait et maîtrisé avant de démarrer ce projet. Les éléments sur le rendu sont en fin de sujet.**

Les noms des fichiers et de certaines fonctions sont imposés et disponibles dans les fichiers main (à compléter) et le `Makefile` (ne pas modifier) sur `CELENE`. Il est impératif de les respecter pour que le projet soit correctement évalué. Il est bien sûr possible d'utiliser des fonctions intermédiaires.

Le projet peut être réalisé sans généricité, en travaillant avec des tableaux de `char *`. Le barème sera divisé par deux dans ce cas.

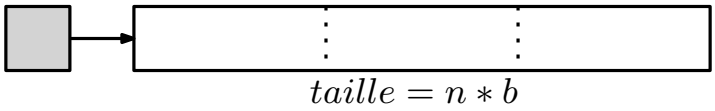
Exercice 1 (8 points)

Tableaux génériques à une dimension (fichiers `exercice1.{h,c}`).

L'objectif est de créer une structure permettant la gestion de tableaux génériques **à une dimension** en `C`, autrement dit de tableaux pouvant contenir *n'importe quel* type de données (chaque case contiendra le **même** type de données). La taille du tableau sera fixée au moment de sa création, et ne sera plus modifiée dans la suite de cet exercice. Le choix des champs de la structure est laissé libre, on notera cependant qu'il est nécessaire de :

- connaître le nombre d'éléments n présents dans le tableau ;
- connaître la taille *mémoire* b du type de données qui sera stocké (pour permettre une gestion correcte de la mémoire et éviter les débordements).

Il n'est pas possible de connaître la taille mémoire associée à un pointeur (sauf cas exceptionnel, comme par exemple les chaînes de caractères). En effet, la fonction `sizeof` utilisée sur un pointeur retourne la taille mémoire occupée par une *variable pointeur* (qui est la même quelque soit le type pointé). Ainsi la connaissance de la taille d'un *bloc* de mémoire ne peut pas permettre de déduire le nombre d'éléments dans un espace mémoire pointé par une variable.



Le rappel

1. Créer une structure `tableau` permettant de gérer des tableaux génériques.
   Le type suivant est imposé : `typedef struct tableau* T;`.
2. Définir une fonction `creer` permettant de créer un tableau générique de n éléments, où la mémoire est réservée mais aucun élément n'est initialisé (sauf éventuellement en mettant les bits à 0).
3. Définir une fonction `afficher` permettant d'afficher les éléments du tableau.
4. Définir une fonction `aleatoire` permettant de créer un tableau de n éléments initialisés aléatoirement. Expliquer dans le rapport la gestion de la génération aléatoire d'un type générique.
5. Définir une fonction `detruire_tout` permettant de libérer **correctement** la mémoire.
6. Définir une fonction `trier` permettant de trier les éléments du tableau générique. *S'inspirer des derniers transparents du `CM #6` présentant la fonction `qsort`.*
7. Tester ces fonctions en utilisant (et en complétant) la fonction principale `main_exercice1.c` déposée sur `CELENE`.

Exercice 2 (6 points)

Tableaux génériques modifiables (fichiers `exercice2.{h,c}`).

On souhaite maintenant enrichir la structure précédente pour pouvoir gérer l'insertion et la suppression d'éléments au sein du tableau. En particulier, la structure contiendra un champ n pour représenter le nombre **maximum** d'éléments que peut contenir le tableau (avant un éventuel ajustement mémoire), et un champ n_{items} pour connaître le nombre d'éléments *réellement* présents dans le tableau.

1. Créer une structure `tableau_variable` afin de pouvoir gérer des tableaux génériques de taille variable. **Le type suivant est imposé : `typedef struct tableau_variable* T_var;`.**
2. Définir une fonction `creer_var` permettant de créer un tableau générique contenant n éléments. Par défaut, le tableau sera vide (et donc $n_{items} = 0$).
3. Définir une fonction `aleatoire_var` permettant de créer un tableau de $p \leqslant n$ éléments initialisés aléatoirement.
4. Définir une fonction `afficher_var` permettant d'afficher les éléments du tableau.
5. Définir une fonction `push` permettant d'insérer un élément **à la fin** du tableau.

   > Lors de l'insertion d'un élément dans le tableau, il faudra s'assurer qu'il reste de la place. Dans le cas contraire, une réallocation mémoire sera nécessaire et la variable n sera modifiée en conséquence.

6. Définir une fonction `pop` retournant **et supprimant** le dernier élément présent dans le tableau.
7. Définir une fonction `detruire_tout` permettant de libérer **correctement** la mémoire.
8. Tester ces fonctions en utilisant (et en complétant) la fonction principale `main_exercice2.c` déposée sur `CELENE`.
9. Définir des fonctions similaires permettant cette fois de supprimer ou d'insérer un élément à un indice donné.

> La gestion actuelle ne réalloue la mémoire que lorsqu'on un ajout d'élément est demandé alors que les n éléments du tableau sont déjà pris. Cette gestion n'est pas réellement pertinente : une série de `pop` va vider le tableau mais la zone mémoire allouée sera toujours la même. Afin de contourner ce problème, on considère un mécanisme simple : lorsque le nombre d'éléments présents dans le tableau n_{items} devient plus petit que la moitié du nombre **maximum** d'éléments n, une réallocation mémoire est réalisée.
>
> Aller plus haut ?

10. Modifier les fonctions de suppression précédentes pour réallouer dynamiquement la mémoire si nécessaire.

Exercice 3 (6 points)

Gestion avancée (fichiers `exercice2.{h,c}`).

L'objectif de cette dernière partie est de réaliser des opérations de gestion plus élaborées sur les tableaux génériques. En particulier, il est demandé de :

1. Définir une fonction `maximum` permettant de retourner l'élément du tableau ayant la valeur **maximum**.
2. Définir une fonction `slice` prenant en paramètres deux entiers $0 \leqslant n \leqslant n_{items}$ et $0 \leqslant m \leqslant n_{items}$ retournant (sous forme de tableau générique) les éléments du tableau contenus entre les indices n inclu et m exclu.
3. Définir une fonction `filtrer` qui doit permettre d'appliquer un *prédicat* `P` sur chaque élément du tableau et de retourner un tableau ne contenant que les éléments pour lesquels `P` retourne true. *Par exemple pour un tableau d'entiers $T = \{1, 2, 3, 4, 5\}$ et le précidat $P(x) = true$ si et seulement si x est pair, la fonction `filtrer(T,P)` doit retourner $\{2, 4\}$.*

Exercice 4

L'application.

Au début de la dernière semaine du projet, un fichier de test plus complet -et plus difficile- sera déposé sur `CELENE`. Il est demandé de l'exécuter et de remplir les parties manquantes. Le barème présenté précédemment sera ajusté en fonction du comportement de votre code par rapport à ce fichier de test.

Le rendu

`git`, PDF et code source.

Le rendu de votre projet se fera **en priorité** via `git`. Un dépôt `CELENE` sera ouvert pour ceux qui en auraient réellement besoin, mais son utilisation entraînera un malus de 1 point.

- Un rapport (maximum 4 pages) au format PDF présentant vos choix d'implémentation et les réponses aux questions explicitement mentionnées dans le sujet.
- Le nom des étudiants doit clairement figurer dans le rapport et dans le nom du fichier. Une répartition du travail est également demandée (en plus des 4 pages autorisées).
- Le code source, avec **uniquement** les fichiers d'en-tête (`.h`), d'implémentation (`.c`) et le `Makefile` permettant de compiler le programme. Les exécutions seront réalisées sur les fichiers de test fournis sur `CELENE`.
- Si le rendu est réalisé via `CELENE`, le contenu du précédent point sera mis dans une archive au format `.zip` (**seul format autorisé**).

Les **seuls** éléments attendus sont listés ci-dessus

Tout projet ne respectant pas les consignes de rendu aura un malus de 2 points sur la note.
Les problèmes de compilation ou de non respect des contraintes seront grandement pénalisés.