TP1 - Tableaux dynamiques

Ce qu'on voit lors de ce TP

- Comment gérer de multiples fichiers dans un projet C
- La notion de tableau dynamique
- Revoir les allocations dynamiques
- Les pointeurs vers des fonctions
- La notion de complexité amortie

I) Comment gérer plusieurs fichiers en C

Peut-être vous ne souvenez plus (ou vous n'avez jamais vu) quelles sont les bonnes habitudes à adopter en C lorsqu'on travaille sur des projets contenant plusieurs fichiers de code. C'est pourtant ce qu'on devra faire pour les TP de ce cours : à chaque structure de données que nous allons coder, on va lui associer ce qu'on appelle un fichier code particulier, ou plutôt ce qu'on appelera un *module*.

L'intérêt est clair : pour pouvoir intégrer facilement ces structures dans d'autres projets, il faut que la partie où on code ces structures soient facilement transférables et donc se fait dans des fichiers à part.

Commencez par télécharger et décompressez l'archive dans un répertoire que vous créerez pour l'occasion sur votre espace personnel.

Sur la page ecampus du cours, vous trouverez les slides du cours que j'enseigne (en L1 !) sur comment s'organise un projet en C qui comporte plusieurs fichiers. Je vous conseille de le lire ou vous documenter directement sur internet si vous préférez. **Vous êtes censés travailler en dehors des TP et en dehors des cours : si vous avez des lacunes, vous devez les combler.** Voici toutefois quelques points sur lesquels je tiens à insister :

Fichier entête

(Quasiment) chaque fichier **.c** s'accompagne d'un fichier **.h** (qu'on appelle aussi *entête* ou *header* en anglais). L'union de ces deux fichiers s'appelle un *module*. Le fichier **.c** est celui qui va contenir le code à proprement parler des fonctions alors que le fichier **.h** est en quelque sorte le mode d'emploi du fichier **.c**. **Ouvrez le fichier `liste.h` et parcourez le document.** Le fichier entête contient traditionnellement de multiples choses :

- la *définition des structures*.

```
/**
 * @brief Implémentation du type tableau dynamique
 *
 */
struct TableauDynamique {

    type_base* tableau;      /**< L'adresse du bloc mémoire où se situe

    size_t taille;      /**< Le nombre actuel d'éléments. */

    size_t capacite; /**< Le nombre maximum d'éléments que peut stocker
    la structure sans faire de réallocation. */

    size_t (*fonction_calcul_capacite) (size_t); /**< Un pointeur vers
    fonction qui indique comment la nouvelle capacité à partir de l'anc
    Ce champ n'a qu'une utilité pédagogique. Normalement on coderait ça

};
```

On reviendra sur le détail de chaque champ de cette structure.

- les *alias* des types.

```
typedef struct TableauDynamique* Liste;
```

Le type de la structure que nous allons manipuler est un pointeur sur la structure définie ci-dessus. Plutôt que d'écrire à chaque fois `struct TableauDynamique*`, on définit ici un alias `Liste` (pour rappeler les *listes* python, à ne pas confondre avec les listes chaînées).

- les *prototypes* des fonctions.

```
size_t longueur(Liste l);
```

Le prototype est juste une déclaration d'une fonction mais sans le code qui va avec. C'est comme quand on écrit `int x;`, on déclare une variable entière mais on l'initialise pas. L'intérêt est double : ça permet aux codeurs (vous en l'occurrence) de connaître les paramètres de la fonction nécessaires pour l'appeler, mais aussi au compilateur : quand on inclut dans un fichier **.h** dans un programme, on recopie juste les prototypes et non l'intégralité des codes des fonctions. A la première étape de la compilation, le compilateur vérifie alors que les fonctions qui sont utilisées dans le programme ont été *a minima* déclarées.

- les *commentaires*.

```
/**
 * @brief La taille de la liste.
 * @param l liste.
 * @returns le nombre d'éléments dans la liste.
 */
```

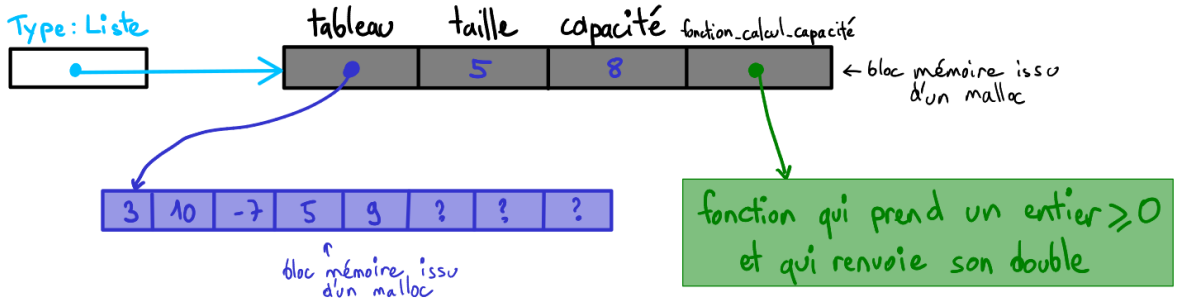
On utilise ici la norme *Doxygen* pour commenter les fonctions. Outre le fait que cette convention est lisible pour l'humain, elle permet de générer automatiquement une documentation de votre projet. **Utilisez la commande `doxygen` pour générer la documentation propre à ce TP.** Cela pourra vous aider pour savoir comment coder les fonctions.

Compiler un projet C

- Quand on veut fabriquer un exécutable à partir de plusieurs fichiers, il faut obligatoirement qu'il y ait une unique fonction `main` parmi tous ces fichiers code. La fonction `main` est celle par laquelle votre programme débutera.
- Toutefois, il est possible de compiler partiellement votre programme. En effet, lorsque vous modifiez un fichier code seul, vous n'êtes pas obligés de recompiler le projet en entier. Il est possible de transformer individuellement chaque fichier code en ce qu'on appelle un *fichier objet*. Un fichier objet c'est juste du code binaire qui contient uniquement ce qui a été écrit dans le fichier **.c**. Il n'est pas exécutable en tant que tel. C'est lors de l'étape de l'*édition des liens* que votre compilateur agrège tous les fichiers objets pour former l'exécutable.
- Pour effectuer ce genre de tâche automatiquement, on utilise la commande `make .` `make` se base sur un fichier appelé `makefile` qui spécifie les règles pour compiler les différents fichiers code. Pour ce TP, vous n'aurez pas à écrire de `makefile` , je vous donne directement ce fichier. Vous pouvez y jeter un coup d'œil. **Retenez-donc que pour compiler/tester le code que vous allez écrire, il faudra juste taper la commande `make` .**

II) Codons la structure de données

Nous allons implémenter la notion de tableau dynamique vue en cours. Voici une représentation schématique de la structure :



Il y a un alias au début de `liste.h` pour définir le type des éléments contenant dans le tableau dynamique. Par défaut, il s'agit d'entiers, mais on pourra modifier cet alias si on souhaite travailler avec autre chose que des entiers.

Les trois premiers champs de la structure sont naturels : `tableau` , l'endroit où se trouve les données, une `taille` et une `capacite` (nombre maximum d'éléments que le tableau dynamique peut a priori accueillir).

La quatrième, `fonction_calcul_capacite` , est non conventionnelle : il s'agit d'un pointeur vers une fonction qui va nous dire comment va grandir notre tableau dynamique à chaque réallocation. D'habitude on code ça sans passer par une fonction auxiliaire. Toutefois, dans ce TP, nous allons jouer avec différentes fonctions de croissances : la présence de ce quatrième champ est donc pédagogique. J'imagine que vous n'avez pas du manipuler beaucoup de pointeurs de fonction, je vais donc faire un rapide laïus dessus.

Pointeurs de fonction ?

Les instructions d'une fonction sont stockées en mémoire : ça veut dire que les fonctions ont une adresse mémoire et donc en toute logique, on peut les manipuler via des pointeurs. Pour déclarer un pointeur de fonctions, la syntaxe "générale" est :

```
type_sortie (*nom_var_ptr_fonction) (type_param1, type_param2, ...);
```

Dans ce TP, le quatrième champ de notre structure est un pointeur vers une fonction qui prend un entier positif et qui renvoie un entier positif, donc on écrit :

```
size_t (*fonction_calcul_capacite) (size_t);
```

Si vous voulez initialiser un pointeur de fonctions, vous pouvez par exemple faire

```
size_t fct (size_t n){
    return n/2;
}
int main() {
    size_t (*ptr) (size_t) = fct;
    printf("%ld\n",ptr(10)); // Affiche 5
    return 0;
```

```
}
```

A noter que normalement, on devrait écrire `&fct` (pour accéder à l'adresse de la fonction) et `(*ptr)(10)`, mais le compilateur convertit tout ça implicitement. Bref, vous pouvez traiter les pointeurs de fonctions comme si vous manipulez directement les fonctions !

Allez on code !

Écrivez dans `liste.c` toutes les fonctions qui sont décrites dans `liste.h`. Attention, il est important de finir toute la liste pour le prochain TP !

Afin de vérifier le bon fonctionnement de vos fonctions, il faudra régulièrement écrire des tests unitaires (`assert` en c) dans la fonction `main` de fichier `main.c`. Pour vous mettre le pied à l'étrier, j'ai commencé à écrire quelques uns de ces tests (en commentaire).

Quelques précisions

- Pour la fonction `liste_vide`, il faudra définir en dehors de la fonction `liste_vide` une fonction qui prend un entier positif et qui renvoie son double afin de mettre son adresse dans le champ `fonction_calcul_capacite`. N'oubliez pas de faire un `malloc` et pour le champ `tableau` et pour la structure.
- Pour la fonction `ajouter_en_fin`, on utilisera la fonction `realloc` dans `stdlib.h` qui permet d'agrandir un bloc mémoire issu d'un `malloc` par la droite si c'est possible. Si par contre l'espace libre est insuffisant, un nouveau bloc sera alloué, les valeurs automatiquement recopiées et l'ancien bloc automatiquement libéré.

III) Benchmark

Dans le fichier `main.c`, **écrivez quatre fonctions** qui prennent en paramètre un entier positif `n` :

- crée une liste vide,
- change le champ `fonction_calcul_capacite` de cette liste vide,
- ajoute `n` fois à la fin de cette liste un entier aléatoire entre 0 et `2*n`,
- libère la mémoire.

Ces quatre fonctions vont différer donc au niveau de la fonction qui calcule les nouvelles capacités :

1. Pour la première fonction, on ne modifiera rien, on garde la fonction qui associe à tout nombre son double.

2. Pour la deuxième fonction, on va utiliser la fonction de croissance utilisé par python (on peut vérifier depuis le code source), à savoir la fonction qui à un entier positif `n` associe

```
((size_t)n + (n >> 3) + 6) & ~(size_t)3
```

3. Pour la troisième fonction, on va utiliser la fonction qui à `n` associe `n+16`.

4. Pour la quatrième fonction, on va utiliser la fonction qui à `n` associe `n+1024`.

(Les deux dernières fonctions rallonge les tableaux dynamiques avec des bouts de tableau de taille constante.)

Grâce au fichier `benchmark`, comparez la vitesse des 4 fonctions pour des tailles 1000, 1 million, 10 million. A votre avis, qui va être la plus rapide ? Complétez le tableau suivant.

taille	double	comme python	ajoute 16	ajoute 1024
mille				
1 million				
10 million				

Selon l'efficacité de la fonction `realloc`, les différences entre les quatre fonctions ne sont peut-être pas si claires. (J'ai eu des résultats différents selon l'heure de la journée.) Si besoin, lancez `valgrind ./test_tableau_dynamique` pour avoir des résultats plus clairs.

Attention, exercice qui peut tomber au CT

Prouvez que quand on rajoute 1024 cases d'entiers à chaque réallocation (plutôt que la taille soit doublée), alors la complexité amortie d'un ajout n'est plus en $O(1)$ mais en $\Theta(n)$, où n est la taille du tableau dynamique. On supposera que le coût d'une réallocation de taille m est en $\Theta(m)$.

(Rappel : $\Theta(n)$ signifie qu'on ne peut pas faire mieux que $O(n)$).

Si vous avez fini ou si vous voulez vous entraîner

Continuez à implémenter les méthodes des listes utilisées dans python : `extend`, `del`, `pop`, la concaténation... Il est susceptible qu'une de ces fonctions tombe au CT.

Pour avoir une liste complète de ces méthodes, allez voir cette page.