Lab #1 - Diverses implémentations de listes.

Objectifs

- Définitions et appels de fonctions manipulant des structures
- Allocation dynamique de mémoire avec `malloc`/`free`
- Manipuler des pointeurs et des tableaux
- Créer de nouveaux types de données
- Programmation modulaire et compilation séparée (`make`)
- Connaître et savoir implémenter différents types de listes

Préliminaires

Vous réaliserez ce travail dans un dépôt Git local. Il vous est demandé de commiter **régulièrement** vos contributions et de *pousser* celles-ci sur la plateforme GitLab (https://gitlab.telecomnancy.univ-lorraine.fr) de l'école. Veillez à organiser votre dépôt de la manière suivante :

- un répertoire `src/` dans lequel vous placerez le code source de votre travail
- ne commiterez que le code source `.c` et non les versions compilées de vos programmes. Pour cela ajouter le nécessaire dans le fichier `.gitignore` pour ignorer les fichiers `.o` notamment.

Rendu

Ce travail est à rendre au plus tard le **lundi 9 mai 2022 à 13:00** sur la plateforme GitLab de l'école.

Pré-requis

Pour l'ensemble du module, nous utiliserons l'environnement de développement intégré Visual Studio Code (https://code.visualstudio.com/) avec les extensions (C/C++ Extension et Makefile Tools) et ainsi que le compilateur Clang (https://clang.llvm.org/). Tous ces outils ont déjà été installés et configurés lors des TPs du module de C.

Vous devez maintenant, si ce n'est pas déjà fait, cloner le dépôt du TP, vous placer dans le répertoire ainsi créer. Plus tard, vous devrez ouvrir ce répertoire dans Visual Studio Code (en utilisant la commande `code .` par exemple).

```
# en SSH
git clone git@gitlab.telecomnancy.univ-lorraine.fr:sd2k22/lab1-oster7.git
# ou en HTTPS (ici on insére l'adresse email dans l'URL lors du "clonage" afin de ne pas
    avoir à la re-saisir plus tard -- à chaque "push")
git clone https://gerald.oster%40telecomnancy.eu@gitlab.telecomnancy.univ-lorraine.fr/
    sd2k22/lab1-oster7.git
```

Ouvrez dans Visual Studio Code le répertoire de votre dépôt. Vous pouvez utiliser la commande suivante :

```
cd lab1-oster7 # si ce n'est déjà fait
code . # pour ouvrir (en tant que projet) le répertoire courant
```

Tests unitaires

Pour ceux d'entre vous qui souhaiteraient aller un peu plus loin que l'objectif de la séance, nous vous invitons à regarder la librairie de tests unitaires Snow afin d'automatiser vos tests. Cette librairie ne nécessite que d'inclure le fichier `snow/snow.h` et de suivre la syntaxe d'usage de la librairie.

Le site web présente un exemple complet (écriture des tests, assertions que vous pouvez utiliser, compilation et exécution).

C'est parti !

Exercices

L'objectif visé pour ces deux premières séances de TPs est de réaliser un outil basique de vérification orthographique. Dans cette optique, vous devrez concevoir et implémenter les structures de données nécessaires. Dans cette séance, vous vous intéresserez à diverses implémentations de la structure liste et aux fonctions s'y rapportant. La séance prochaine sera dédiée à l'implémentation d'une table de hachage et au dernier élément – le vérificateur qui devra charger un fichier texte et vérifier si chaque mot le composant est présent, ou non, dans le dictionnaire.

Exercice 1 - Liste simplement chaînée

L'objectif de cet exercice est d'écrire une liste simplement chaînée permettant de conserver des éléments de type valeur entière (`int`). Vous réaliserez ensuite des implémentations permettant d'y stocker une chaîne de caractères (`char *`) et une structure de données quelconque (`element_t *`).

Question 1.

Écrivez l'implémentation d'une liste simplement chaînée de valeurs entières (`int`) en découplant l'interface et l'implémentation dans les fichiers `linked_list_int.h` et `linked_list_int.c`. Vous réaliserez également des tests unitaires validant votre implémentation sous la forme d'un fichier `linked_list_int_test.c` permettant de générer un exécutable.

La structure attendue portera le nom `linked_list_int_t`.

Les fonctions attendues doivent permettre de :

- créer une liste : `linked_list_int_t* list_create()`
- vérifier si une liste est vide : `bool list_is_empty(linked_list_int_t* one_list)`
- ajouter un élément en tête de liste : `void list_prepend(linked_list_int_t* one_list, int one_value)`
- consulter le premier élément de la liste : `int list_first(linked_list_int_t* one_list)`
- ajouter un élément en queue de liste : `void list_append(linked_list_int_t* one_list, int one_value)`
- consulter le dernier élément de la liste : `int list_first(linked_list_int_t* one_list)`
- visualiser une liste en l'affichant sur la sortie standard : `void list_print(linked_list_int_t* one_list )` selon le format suivant `[ 1, 2, 3, 4 ]`

— insérer un élément en n-ième position (par convention les positions débuterons à 0) à la liste : `void list_insert(linked_list_int_t* one_list, int one_value, unsigned int index)`
— consulter le i-ième élément (celui situé à la position `i`) de la liste `int list_get(linked_list_int_t* one_list, unsigned int index)`
— recherche l'indice d'un élément (l'indice de la première occurrence de l'élément) de la liste `unsigned int list_index_of(linked_list_int_t* one_list, int one_value)`
— détruire une liste : `void list_destroy(linked_list_int_t* one_list)`

Question 2.

Écrivez l'implémentation d'une liste simplement chaînée de chaînes de caractères (`char *`) en découplant l'interface et l'implémentation dans les fichiers `linked_list_string.h` et `linked_list_string.c`. Vous réaliserez également des tests unitaires validant votre implémentation sous la forme d'un fichier `linked_list_string_test.c` permettant de générer un exécutable.

La structure attendue portera le nom `linked_list_string_t`.

Les fonctions attendues sont les mêmes que pour la question précédente. Vous adapterez uniquement le type de l'élément conservé ou de la valeur de retour.

Question 3.

Écrivez l'implémentation d'une liste simplement chaînée d'éléments d'une structure quelconque (`element_t *`) en découplant l'interface et l'implémentation dans les fichiers `linked_list.h` et `linked_list.c`. Vous réaliserez également des tests unitaires validant votre implémentation sous la forme d'un fichier `linked_list_test.c` permettant de générer un exécutable.

La structure de l'élément conservée portera le nom `element_t` et la structure de la liste portera le nom `linked_list_t`,

Les fonctions attendues sont les mêmes que pour la question précédente. Vous adapterez uniquement le type de l'élément conservé ou de la valeur de retour.

Question 4.

De quelle manière, serait-il possible de « factoriser » le code des 3 questions précédentes pour réaliser une implémentation « générique » ?

Exercice 2 - Liste doublement chaînée

L'objectif de cet exercice est d'écrire une liste doublement chaînée permettant de conserver des éléments d'une structure quelconque (`element_t *`).

Question 5.

Écrivez l'implémentation d'une liste doublement chaînée d'éléments d'une structure quelconque (`element_t *`) en découplant l'interface et l'implémentation dans les fichiers `double_linked_list.h` et `double_linked_list.c`. Vous réaliserez également des tests unitaires validant votre implémentation sous la forme d'un fichier `double_linked_list_test.c` permettant de générer un exécutable.

La structure de l'élément conservée portera le nom `element_t` et la structure de la liste portera le nom `double_linked_list_t`,

Les fonctions attendues sont les mêmes que pour les questions précédentes. Vous adapterez uniquement le type de l'élément conservé ou de la valeur de retour.

Exercice 3 - Liste contiguë

L'objectif de cet exercice est d'écrire une liste contiguë permettant de conserver des éléments d'une structure quelconque (`element_t *`). Ainsi, les éléments de la liste sont stockées dans des zones contiguës de la mémoire et il est possible d'accéder à un élément donné (en connaissant son index) en temps constant.

Question 6.

Écrivez l'implémentation d'une liste contiguë d'une structure quelconque (`element_t *`) en découplant l'interface et l'implémentation dans les fichiers `contiguous_list.h` et `contiguous_list.c`. Vous réaliserez également des tests unitaires validant votre implémentation sous la forme d'un fichier `contiguous_list_test.c` permettant de générer un exécutable.

La structure de l'élément conservée portera le nom `element_t` et la structure de la liste portera le nom `contiguous_list_t`,

Les fonctions attendues sont les mêmes que pour les questions précédentes. Vous adapterez uniquement le type de l'élément conservé ou de la valeur de retour. La fonction de création de la liste permettra d'indiquer la capacité initiale (cette capacité est amenée à augmenter lors de l'utilisation) de la liste : `contiguous_list_t* list_create(unsigned int capacity)`.

Question 7.

Écrivez l'implémentation d'une liste contiguë circulaire d'une structure quelconque (`element_t *`) en découplant l'interface et l'implémentation dans les fichiers `circular_contiguous_list.h` et `circular_contiguous_list.c`. Vous réaliserez également des tests unitaires validant votre implémentation sous la forme d'un fichier `circular_contiguous_list_test.c` permettant de générer un exécutable.

La structure de l'élément conservée portera le nom `element_t` et la structure de la liste portera le nom `circular_contiguous_list_t`,

Les fonctions attendues sont les mêmes que pour les questions précédentes. Vous adapterez uniquement le type de l'élément conservé ou de la valeur de retour.

Exercice 4 - Évaluation des performances

Question 8. Évaluez les performances des différentes implémentations que vous avez réalisées en effectuant des mesures empiriques.