P31 Développement efficace - épisode 1

Dans le cadre de ce module, nous allons mettre en oeuvre des structures de données classiques et quelques algorithmes qui les utilisent.

Le langage de programmation que nous allons utiliser est le C.

---

1 - Compilation / Exécution / Organisation du code

Dans le dossier `p31` fourni, vous trouvez des fichiers source (`.h` et `.c`) ainsi qu'un `Makefile`.

Le code fourni compile en l'état. À l'exécution de la commande `make`, deux dossiers sont créés :

- `build` : contient les fichiers `.o` produits par la compilation de chaque fichier `.c`
- `bin` : contient l'exécutable `p31`

Pour exécuter le programme, vous pouvez lancer la commande `./bin/p31`.

> ***Compiler et exécuter le programme.***

Bien entendu, vous devez modifier la fonction `main` au fil des exercices afin de tester l'ensemble des fonctionnalités développées.

Le code sera réparti dans 2 dossiers :

- `containers` : contient les fichiers relatifs à la définition des structures de données
- `algos` : contient les fichiers relatifs aux algorithmes

---

2 - Un `vector` d'entiers

Un `vector` permet de stocker de manière contiguë en mémoire un ensemble d'éléments de même taille. Les éléments sont accessibles par leur indice dans le conteneur. La taille du `vector` est modifiée dynamiquement au fur et à mesure de l'ajout d'éléments.

Afin de ne pas réallouer la mémoire à chaque ajout d'élément, un `vector` met en place un mécanisme d'allocation anticipée : la zone mémoire est initialisée avec une taille donnée, puis à chaque ajout d'élément, s'il reste de la place, on stocke le nouvel élément, s'il n'y a plus de place, on commence par allouer une nouvelle zone plus grande dans laquelle on copie le contenu de la zone initiale avant de la libérer, puis on stocke le nouvel élément.

Pour cela, on aura besoin de connaître 3 choses à tout moment :

- `data` : l'adresse de la zone mémoire allouée
- `capacity` : la capacité de la zone mémoire allouée (en nombre d'éléments)
- `size` : la taille courante, ou nombre d'éléments effectifs (forcément inférieur ou égal à la capacité)

> ***Compléter la déclaration du type `vector_int` dans le fichier `containers/vector_int.h`, pour mettre en oeuvre un `vector` d'entiers.***

Dans le fichier `containers/vector_int.h`, des prototypes de fonctions sont déjà déclarés. Les corps de toutes ces fonctions sont écrits (mais vides) dans le fichier `containers/vector_int.c`. Des commentaires sont donnés dans le code afin de détailler ce qui est attendu.

> ***Écrire le corps de toutes ces fonctions.***

Une fois ces fonctions écrites, on pourra écrire par exemple :

```c
vector_int v; // déclaration
vec_int_init(&v); // initialisation
for (size_t i = 0; i < 20; i++)
    vec_int_push_back(&v, rand_int_between(0, 100));
printf("vector_int content :\n");
for (size_t i = 0; i < vec_int_size(&v); i++)
    printf("%d - ", vec_int_get_value(&v, i));
printf("\n");
vec_int_free(&v); // libération
```

La fonction `rand_int_between` est fournie dans les fichiers `utils.h` et `utils.c`.

---

3 - Premier algorithme

Dans le fichier `algos/bubble_sort.h`, un prototype de fonction est déjà déclaré.

> ***Dans le fichiers `algos/bubble_sort.c`, écrire le corps de la fonction correspondante.***

Dans votre `main`, tester le bon fonctionnement de cette fonction en affichant le contenu d'un `vector_int` avant et après appel à la fonction de tri.

Pour mesurer le temps mis par l'exécution de la fonction de tri, écrire quelque chose comme :

```c
srand(time(NULL));
struct timeval tval_before, tval_after, tval_result;

// initialisation d'un vector v
// ...
// affichage de son contenu initial
// ...

gettimeofday(&tval_before, NULL);

bubble_sort_vec_int(&v);

gettimeofday(&tval_after, NULL);
timeval_sub(&tval_after, &tval_before, &tval_result);
printf("\tElapsed time: %ld.%06ld\n",
  (long int)tval_result.tv_sec, (long int)tval_result.tv_usec);

// affichage de son contenu après tri
// ...
// libération du vector v
// ...
```