# Structures de données
## TP3

# Introduction

Dans le TP précédant, vous avez développez en C un implémentation simple de la classe C++ « `Vector` » (documentation ici) permettant de stocker uniquement des `double`.

Dans ce TP, vous allez développer des testes pour évaluer les performances, en temps et en mémoire, de votre implémentation. Puis vous apporterez une amélioration à votre tableau dynamique.

# 1 Évaluation des performances

Pour évaluer les performances d'un tableau dynamique, vous mesurerez :
- le temps d'exécution ;
- le nombres d'allocation mémoire ;
- la quantité moyenne de mémoire alloué.

Dans les scénarios suivant :
- l'ajout et la suppression d'éléments à des indexes aléatoire ;
- l'ajout et la suppression d'éléments en tête et en queue ;
- l'accès aléatoires en lecture et écriture ;
- l'accès séquentielle en lecture et écriture ;
- tri des données

## 1.1 Générateur aléatoire

Dans une premier temps, vous allez développer un ensemble de fonctions pour générer des nombres et chaîne aléatoire.

1.1. Créez dans votre répertoire de travail les fichiers
- « `random.h` » le fichier d'entête de vos fonctions de génération aléatoire ;
- « `random.c` » le fichier de code de vos fonctions de génération aléatoire ;
- « `test_random.c` » le fichier de testes unitaires de vos fonctions (contient le `main`).

(a) Ajoutez dans le fichier « `random.h` », les instruction de précompilation pour sécuriser votre fichier contre les doubles inclusions.

(b) Ajoutez dans les fichiers « `random.c` » et « `test_random.c` », l'inclusion du fichier « `random.h` ».

(c) Ajoutez dans le fichier « `test_random.c` », la fonction `main`.

(d) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `random.c` » et obtenir le fichier « `random.o` ».

(e) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `test_random.c` » et obtenir le fichier « `test_random.o` ».

(f) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `test_random.c` » et obtenir le fichier « `test_random.o` ».

(g) Ajoutez dans le fichier « `makefile` », les lignes pour compiler l'exécutable final « `test_random` » a partir des fichiers « `test_random.o` » et « `random.o` ».

(h) Testez en exécutant la commande `make test_random` dans votre répertoire de travail.

1.2. Générateur de nombre aléatoire En C, la fonction permettant de générer des nombres aléatoires est la fonction `int rand(void);`, elle est défini dans `#include <stdlib.h>`. Cette fonction génère des nombres pseudo-aléatoire, à chaque appel, elle retourne un nombre entier compris entre 0 et `RAND_MAX` (inclus).

Le générateur de nombre étant pseudo-aléatoire, il fournie à chaque exécution la même séquence de nombres aléatoires. Pour éviter cela, il est courant d'initialiser la grène du générateur de nombre aléatoire avec la date lors de l'exécution du programme. Pour cela, vous utiliserez les des fonction suivante :

— `void srand(int start)` (documentation ici)
— `int time(int*)` (documentation ici)

(a) Écrivez la fonction `double random_double(double a, double b);` qui retourne un nombre aléatoire de type `double` compris entre `a` et `b` exclu. Écrivez un code de test dans « `test_random.c` ».

(b) Écrivez la fonction `float random_float(float a, float b);` qui retourne un nombre aléatoire de type `float` compris entre `a` et `b` exclu. Écrivez un code de test dans « `test_random.c` ».

(c) Écrivez la fonction `size_t random_size_t(size_t a, size_t b);` qui retourne un nombre aléatoire de type `size_t` compris entre `a` et `b` exclu. Écrivez un code de test dans « `test_random.c` ».

(d) Écrivez la fonction `int random_int(int a, int b);` qui retourne un nombre aléatoire de type `int` compris entre `a` et `b` exclu. Écrivez un code de test dans « `test_random.c` ».

(e) Écrivez la fonction `unsigned char random_uchar(unsigned char a, unsigned char b);` qui retourne un nombre aléatoire de type `unsigned char` compris entre `a` et `b` exclu. Écrivez un code de test dans « `test_random.c` ».

(f) Écrivez la fonction `void random_init_string(unsigned char * c, size_t n);` qui remplit la chaîne de caractère `c` de lettre majuscule aléatoire,`n` est la taille du tableau pointé par le pointeur `c`. Écrivez un code de test dans « `test_random.c` ».

## 1.2 Test de votre vecteur dynamique

Maintenant, vous allez créer un programme pour évaluer les performances de votre vecteur dynamique.

Ce programme prendra en paramètre, le type de test à exécuter et le nombre de test à exécuter, avec la structure suivante :

```
./bench_vector_v1_double test_type init_size n
```

avec `test_type` le type du teste à exécuter, `init_size` la taille initial du tableau dynamique et `n` le nombre d'exécution du test. Pour cela vous devrez utiliser les arguments `argc` et `argv` de la fonction `int main(int argv, char *argv[])`

1.1. Créez dans votre répertoire de travail le fichier : « `bench_vector_v1_double.c` »

(a) Ajoutez dans le fichier « `bench_vector_v1_double.c` », la fonction `main`.

(b) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `bench_vector_v1_double.c` » obtenir le fichier « `bench_vector_v1_double.o` ».

(c) Ajoutez dans le fichier « `makefile` », les lignes pour compiler l'exécutable final « `bench_vector_v1_do` partir des fichiers « `bench_vector_v1_double.o` », « `vector_v1_double.o` » et « `random.o` ».

(d) Testez en exécutant la commande `make test_random` dans votre répertoire de travail.

1.2. Les fonctions pour les différents tests.

(a) Écrivez dans votre fonction `main`, le code pour récupérer les arguments passé au programme et convertissez les arguments `init_size` et `n` en valeur numérique de type `size_t`. (pensez à utilisé la fonction `int sscanf(const char *str, const char *format, ...);` documentation ici.

(b) Après avoir récupérez les paramètres, écrivez dans votre fonction `main`, le code qui alloue un `vector_v1_double` de taille `init_size` et qui l'initialise aléatoirement.

(c) Écrivez la fonction qui `void insert_erase_random(p_s_vector_v1_double p_vector, size_t n );` qui répète `n`-fois l'ajout puis la suppression d'un élément à des positions aléatoire. Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égale à "insert_erase_random".

(d) Écrivez la fonction qui `void insert_erase_head(p_s_vector_v1_double p_vector, size_t n);` qui répète `n`-fois l'ajout puis la suppression d'un élément en tête. Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égale à "insert_erase_head".

(e) Écrivez la fonction qui `void insert_erase_tail(p_s_vector_v1_double p_vector, size_t n );` qui répète `n`-fois l'ajout puis la suppression d'un élément en queue. Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égale à "insert_erase_tail".

(f) Écrivez la fonction qui `void read_write_random(p_s_vector_v1_double p_vector, size_t n);` qui répète `n`-fois l'écriture puis la lecture d'un élément a des positions aléatoire. Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égale à "read_write_random".

(g) Écrivez la fonction qui `void read_write_sequential(p_s_vector_v1_double p_vector, size_t n);` qui répète `n`-fois l'écriture de tous les éléments du vecteur puis la lecture de tous les éléments du vecteur toujours avec un parcours de la tête vers la queue. Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égale à "read_write_sequential".

(h) Écrivez la fonction qui `void bubble_sort(p_s_vector_v1_double p_vector, size_t n);` qui répète `n`-fois l'écriture de tous les éléments du vecteur avec des valeur aléatoire puis tri du vecteur avec le tri à bulles (documentation ici) Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égale à "bubble_sort".

1.3. Test des performance de votre structure pour les différents cas de test et différente valeur de `init_size` et `n`.

(a) Utilisez la commande Linux `time` (documentation ici) pour mesurer le temps d'exécution de votre `bench_vector_v1_double`.

(b) Utilisez la commande Linux `valgrind` (documentation ici) pour mesurer le nombre d'allocation et la quantité total de mémoire alloué lors de l'exécution de votre `bench_vector_v1_double`.

# 2 Tableau dynamique V2.0

Maintenant, vous allez ajouter à votre structure de tableau dynamique, une stratégie pour éviter de ré-allouer le tableau à chaque insertion ou suppression d'un élément. Pour cela, nous allons utilisé deux variable :

— **size** : le nombre d'éléments « référencés » dans la structure ;

— **capacity** : le nombre d'éléments « référençables » dans la structure (taille réel du tableau alloué).

Les règles que nous allons suivante sont les suivantes :

1. La capacité minimum de notre structure est de 16 éléments (capacity $\geq 16$.) ;
2. Après l'ajout d'un élément, si capacity = size alors vous doublez la capacité de votre structure (capacity $\leftarrow 2 \times$ capacity) ;
3. Après la suppression d'un élément, si size $\leq \frac{\text{capacity}}{4}$ alors vous divisez par 2 la capacité de votre structure (capacity $\leftarrow \frac{\text{capacity}}{2}$)

2.1. Créez une copie de tout les fichiers de votre structure de données `vector_v1_double` (`vector_v1_double.c` `vector_v1_double.h`, `test_vector_v1_double.c`, ...) et renommez les en `vector_v2_double`.

2.2. Ajoutez dans le fichier « `makefile` », les lignes pour compiler l'ensemble des fichiers de `vector_v2_double`.

2.3. Dans `vector_v2_double`, modifiez votre structure pour votre tableau dynamique en y ajoutant une variable pour stocker la capacité.

2.4. Dans `vector_v2_double`, modifiez les fonctions de votre tableau dynamique pour prendre en compte les règles pour limité le nombre de réallocation.

2.5. Re-testez et regardé les performances après vos modification.