T.P. 4

Une exploration de la bibliothèque standard

fencour Ce TP est une revue assez sommaire de bibliothèques importantes incluses dans la bibliothèque standard. Certaines d'entre elles n'ont été introduites qu'à partir du standard C++11 mais sont déjà bien ancrées dans la tradition.

Le TP1 vous a permis de découvrir brièvement `<vector>` et une partie de la bibliothèque `<algorithm>`. N'hésitez pas à (ré)utiliser autant que possible cette dernière dans ce TP.

4.1 Exercice avec `<random>` : un exemple d'héritage avec la marche aléatoire

Soit la marche aléatoire simple $(S_n)_{n\geq 0}$ définie pour tout $n \geq 0$ par $S_n = \sum_{k=1}^{n} X_k$ où les X_k sont des v.a. indépendantes telles que $\mathbb{P}(X_k = 1) = p$ et $\mathbb{P}(X_k = -1) = 1-p$. Nous souhaitons réaliser une simulation de cette marche aléatoire et du processus $(M_n)_{n\in\mathbb{N}}$ défini par $M_n = \min(S_0, \ldots, S_n)$.

Pour cela, vous utiliserez la bibliothèque `<random>` et un générateur de nombres pseudo-aléatoires de type `std::mt19937`. Une documentation de la classe est disponible à l'adresse :

https://en.cppreference.com/w/cpp/numeric/random

Question 4.1. Écrire une courte classe

```
class RandomWalk {
   protected:
      unsigned n;//temps courant n
      int s;// valeur de S_n
      int s_init;//valeur de S_0
      std::bernoulli_distribution U;//paramètre p
   public:
      RandomWalk(int s0,double p);
      ... val(...) ...  //accesseur à s
      ... time(...) ...  //accesseur à n;
      ... reset() ... // redémarrage à l'état initial
```

```
12      template .... void update( ... G) ... ;
            // mise à jour de s: passage de n à n+1 un générateur G de type arbitraire
14  };
```

pour implémenter l'évolution de la marche aléatoire $(S_n)_{n\in\mathbb{N}}$.

Question 4.2. Tester votre classe avec un court programme qui affiche plusieurs réalisations des 10 premiers pas d'une marche.

Question 4.3. Écrire une courte classe

```
class RandomWalk_with_Min: public RandomWalk {
2   protected:
       int m;//valeur de M_n
4   public:
       RandomWalk_with_Min(int s0,double p): ...
6      ... minimum() ...; //accesseur à m
       ...//compléter/modifier les méth. de la classe mère si nécessaire.
8   };
```

qui hérite de la précédente et ajoute le calcul du minimum absolu courant (M_n). *Aucune modification de* `RandomWalk` *ne doit être faite et aucune répétition de code par rapport à* `RandomWalk` *ne doit être présente.*

Question 4.4. Écrire un programme `RW.cpp` qui utilise la classe précédente pour réaliser une simulation de longueur $T = 10000$ et écrit les valeurs successives de $(S_n)_{0\leq n<T}$ dans un fichier `RW.dat` et celles de $(M_n)_{0\leq n<T}$ dans un fichier `RWmin.dat` , en mettant sur chaque ligne le temps n, la valeur de S_n et celle de M_n, toutes séparées par des espaces.

Question 4.5. Visualisez les fichiers avec `gnuplot` en tapant dans ce dernier :

```
plot "RW.dat" with lines, "RWmin.dat" with lines
```

Relancer plusieurs fois le programme et observer le changement graphique. Cela vous semble-t-il cohérent ?

4.2 Exercice sur conteneurs et itérateurs : mesure de performance

4.2.1 Rappels

Le mot *conteneur* désigne en C++ une classe (ou plus précisément un *template* de classe) qui permet de décrire une collection d'objets de même type. Selon le *conteneur*, cette collection peut être ordonnée ou non et le codage en mémoire et les complexités peuvent différer. Plusieurs opérations essentielles sont communes à tous les conteneurs :

1. compter le nombre d'éléments du conteneur (`size()`),
2. effacer tout le conteneur (`clear()`),
3. ajouter ou enlever un élément (éventuellement à un endroit précis si le conteneur est ordonné) : selon l'endroit et/ou le conteneur, `insert` , `erase` , `pop_front` (enlever au début), `pop_back` (enlever à la fin), `push_front` (ajouter au début), `push_back` (ajouter à la fin)
4. la possibilité de le parcourir entièrement sans oublier d'éléments (de manière ordonnée si le conteneur l'est, aléatoirement sinon) via

```
for(auto x : conteneur) //ou bien const auto & ou bien auto &
```

   ou toute fonction de `<algorithm>` via les itérateurs de début et de fin,
5. trouver si un élément est présent et récupérer l'emplacement où il est stocké (`find`)

Nous vous encourageons à consulter l'immense tableau de la page

https://en.cppreference.com/w/cpp/container

pour voir quelles opérations sont disponibles sur quels conteneurs.

Les deux derniers points de la liste ci-dessus sont gérés par le concept d'*itérateur*. Un *itérateur* est un objet qui permet :

— de pointer un emplacement d'un conteneur
— de se déplacer intelligemment dans un conteneur.

Étant donné un itérateur `it` , on peut :

— accéder à la valeur sur l'emplacement indiqué par `*it` ,
— passer à l'élément suivant par `it++` ou `++it` ,

Chaque itérateur existe sous une version en lecture et écriture et une version en lecture seule. Les prototypes sont donnés par

```
std::CONTENEUR<TYPE>::iterator
std::CONTENEUR<TYPE>::const_iterator
```

Chaque conteneur `X` possède deux méthodes

— `X.begin()` qui pointe vers son premier élément (`X.cbegin()` pour la version en lecture seule),
— `X.end()` qui pointe vers un élément fantôme indiquant qu'on a déjà franchi le dernier élément (`X.cend()` pour la version en lecture seule),

Remarque : vous avez déjà utilisé les itérateurs à chaque fois que vous avez utilisé une fonction de `<algorithm>` .

4.2.2 Performances sur différents conteneurs ordonnés

Question 4.6. (performance de quelques algorithmes sur les vecteurs) Écrire une fonction

```
void measure_complexity_on_vector(long int N)
```

qui fait les étapes suivantes :

- crée deux vecteurs `c1` et `c2` de `double` de taille N,
- remplit le premier avec des nombres aléatoires indépendants de loi exponentielle de paramètre 1,
- compte le nombre d'éléments supérieurs à 10 dans le premier,
- élève au carré les 100 premiers éléments du premier,
- copie le premier dans le deuxième,
- trie la première moitié du premier avec `std::sort`
- échange le contenu des deux vecteurs via `std::swap` .
- écrit le deuxième dans un fichier `chocolatine.cpp`

Pour chaque étape, vous chronométrerez (avec une procédure similaire à ce qui a été fait dans le TP2 en utilisant `<chrono>`) le temps de calcul nécessaire et l'afficherez dans la console.

Question 4.7. Utilisez la fonction pour $N = M = 10000000$ (dix millions), puis $N = 2M$, $N = 4M$ et $N = 8M$. **Discuter** l'évolution des temps obtenus le plus précisément possible..

Question 4.8. Faire une deuxième fonction en utilisant `std::deque` à la place de `std::vector` . **Discuter** les différences de temps avec l'exemple précédent.

Question 4.9. Faire une troisième fonction en utilisant `std::list` à la place `std::vector` *sans faire l'étape de tri.* **Discuter** les différences de temps avec les exemples précédents.

Question 4.10. (bonus) Si vous avez utilisé la bibliothèque `<algorithm>` , refaites le programme en écrivant toutes les boucles à la main (sauf pour le tri) et comparez les temps obtenus. Si vous avez tout fait avec des boucles, refaites le programme avec `<algorithm>` et comparez les temps obtenus.

4.3 Exercice sur le conteneur `std::map` : analyse d'un texte

Le conteneur `std::map` (ainsi que `std::unordered_map`) fournit l'équivalent C++ des dictionnaires de Python : il permet de stocker l'association d'une valeur à une autre valeur. Une documentation restreinte est disponible dans le polycopié de cours et la documentation générale avec maints exemples est disponible à l'adresse

https://en.cppreference.com/w/cpp/container/map

Question 4.11. Le fichier `declaration.txt` contient le préambule de la déclaration des droits de l'homme en français, sans accent ni ponctuation ni majuscule. Écrire un programme `texte_analyse.cpp` qui fait les choses suivantes :

- ouvre le fichier `declaration.txt` en lecture,
- déclare un objet `S` de type `std::map<std::string,unsigned>` vide,
- lit le texte et, à chaque mot, incrémente `S` afin qu'à la fin `S[mot]` indique le nombre de fois que `mot` apparaît dans le texte.

Dans toutes les question suivent de cette section sauf la dernière, on complètera ce programme. Les questions avec une (*) sont plus difficiles

Question 4.12. Écrire dans un fichier `stats.dat` chaque mot avec sa fréquence en ordre lexicographique.

Question 4.13. (bonus) Écrire dans un fichier `stats2.dat` chaque mot avec sa fréquence par ordre décroissant de fréquence.

Question 4.14. Répondre numériquement aux questions suivantes :

1. Combien y a-t-il de mots différents ?
2. Combien y a-t-il de mots différents de plus de 7 lettres ou plus ?
3. Quel est le mot le plus fréquent et combien de fois apparaît-il ? Donnez-en un. Bonus : donnez les tous.
4. Combien y a-t-il de lettres au total dans la déclaration ?
5. (*) Combien y a-t-il de mots avec exactement deux voyelles ?
6. (*) Quels mots contiennent le plus de fois la lettre 'e' ?
7. (*) Quelle est la corrélation empirique entre la longueur d'un mot et sa fréquence ?

Question 4.15. (*) Afficher tous les mots qui ont plus de 12 lettres, apparaissent au moins 13 fois, ne contiennent pas la lettre "e" et ne terminent pas par la lettre "s".

Question 4.16. (bonus) Écrire une fonction

```
std::set<std::string> filter(
2       const std::map<string,unsigned> & M,
       char first_letter, unsigned k);
```

qui construit l'ensemble des mots qui commencent par `first_letter` et apparaissent au moins `k` fois.

Application : écrire dans un fichier `h2.dat` l'ensemble des mots qui commencent par "h" et apparaissent au moins 2 fois.

Un retour sur la première section de cette feuille.

Question 4.17. (bonus+) Reprendre la marche aléatoire de la section 4.1 et, à l'aide de `std::map<int,unsigned>` , réaliser l'histogramme de la position finale de la marche (S_n) pour $n = 10000$ et un grand nombre de réalisations indépendantes ($M = 10^7$) de la marche et l'écrire dans des fichiers Visualiser avec l'outil de votre choix (gnuplot, matplotlib, etc). Quel théorème de probabilité cela illustre-t-il ?

4.4 Exercice : les retours multiples avec `<tuple>`

L'une des grandes frustrations du langage C et du langage C++ jusqu'au standard C++03 est l'impossibilité pour une fonction de renvoyer plusieurs objets. Pour contrer cela, il fallait soit renvoyer des structures définies *ad hoc*, soit passer les variables de sorties en argument via des pointeurs ou des références (ce qui rend la lecture des prototypes difficile sans une documentation correcte). L'introduction des `std::tuple` dans la bibliothèque `<tuple>` à partir du standard C++11 permet de contourner cela.

Mode d'emploi : un objet déclaré selon

```
std::tuple<double, int, std::string> X;
```

contient trois objets auquel on accède en lecture comme en écriture par `std::get<0>(U)` , `std::get<1>(U)` et `std::get<2>(U)` . On a droit aux opérations suivantes :

```
double x;
int n;
std::string s;
...
auto X=std::make_tuple(x,n,s); // définition de X à partir de x, n, s
...
std::tie(x,n,s)=X; // et réciproquement remplissage de x,n,s à partir de X
...
auto [y,m,ss]=X; // définition et initialisation de y,m,ss à partir de X
                 // (uniquement à partir du standard c++17)
```

Question 4.18. Reprendre l'exemple de la section précédente avec le programme intitulé `texte_analyse.cpp` et le compléter en écrivant une fonction :

```
std::tuple< double, unsigned, std::vector<std::string>,  >
basic_statistics(const std::map<std::string,unsigned> & M);
```

qui renvoie simultanément le nombre moyen de lettres par mots (pondéré par la fréquence des mots), le nombre de lettres du mot le plus long et la liste des mots les plus longs.

La tester ensuite et afficher le résultat.