Évaluation du TP : Le travail est à faire en binôme et sera évalué.

# Espace de noms

En C++, on peut « ranger » ses déclarations de variables, de constantes, de types et de fonctions (et les définitions associées) dans un espace de noms :

```
// MonComposant.hpp : déclarations des variables, constantes,
// classes et fonctions
namespace MonEspaceDeNom
{ // les symboles déclarés ici appartiennent à l'espace de nom
  // MonEspaceDeNom
      int n ;
      const float PI = 3.14159 ;

      class MaClasse
      {
           …
      } ;

      void MaFonction() ;
} // pas de point-virgule ici !!

// MonComposant.cpp : définition des classes et des fonctions
namespace MonEspaceDeNom
{
      void MaClasse::maMethode( … )
      {
           …
      }
      …
      void MaFonction()
      {
          …
      }
} // pas de point-virgule ici !!
```

Un espace de noms ne peut être défini qu'à un niveau global, c'est-à-dire en dehors de toute fonction et de tout type structuré (structure, classe).
Un espace de noms peut être créé/construit *incrémentalement* et cela dans plusieurs fichiers sources différents. Cela permet de « ranger » des classes distinctes dans le même espace de nom, simplement en entourant les déclarations et définitions de **namespace** MonEspaceDeNom { … }.

Ensuite pour utiliser nos types et fonctions déclarés dans l'espace de noms `MonEspaceDeNom` :

```
using namespace MonEspaceDeNom ;
int main()
{
    MaClasse c(…) ;
    …
}
```

La déclaration de **using** doit suivre l'inclusion d'un include qui définit au moins partiellement l'espace de nom en question pour que ce dernier soit connu.
La déclaration de **using** peut être locale à un bloc { … } afin de choisir des symboles particuliers uniquement dans une fonction particulière.
La déclaration de **using** peut être spécifique à un symbole d'un espace de noms et pas à l'espace de noms tout entier : **`using`** `MonEspaceDeNom::MaClasse`. `MaClasse` sera alors synonyme de `MonEspaceDeNom::MaClasse`.

**Les programmes de tests demandés dans les questions suivantes doivent se trouver au sein de fonctions déclarées et définies dans un espace de noms.** De plus, si vous déclarez des types pour vos tests, vous devrez aussi les ranger dans un espace de nom. **Votre espace de noms sera Nom1_Nom2 où Nomi est le nom de famille du membre i du binôme.**

# STL

Les conteneurs de la **STL** (*Standard Template Library*) sont des conteneurs à utiliser prioritairement par rapport à tout conteneur « maison ». **Chaque conteneur de la STL est un patron de classes (classe générique), dont le type de l'élément à stocker est le 1er paramètre du patron**, les autres paramètres du patron ayant tous une valeur par défaut. Chaque conteneur de la STL a une taille dynamique, susceptible de varier pendant l'exécution.

## Étude des **conteneurs séquentiels**, de leurs itérateurs et des algorithmes existants sur les conteneurs séquentiels

Dans un conteneur séquentiel, les éléments stockés sont ordonnés (il y a un début et une fin) et on peut parcourir le conteneur suivant cet ordre. Il y a 2 familles de conteneurs séquentiels, les **tableaux dynamiques** (bloc(s) contiguës de mémoire) et les **listes** (pas de contiguïté physique).

- Pour les tableaux dynamiques l'accès à un élément est direct/en temps constant, tandis que pour les listes l'accès à un élément est linéaire en le nombre d'éléments.
- Pour les listes doublement chaînées la suppression ou l'insertion d'un élément à un endroit aléatoire est en temps constant pendant un parcours, tandis qu'elles sont linéaires en le nombre d'éléments pour les tableaux dynamiques ou les listes simplement chaînées.

**Pour choisir entre un tableau dynamique et une liste, il faut se demander si l'on doit faire beaucoup d'accès aléatoire, ou beaucoup d'insertion/suppression à un endroit connu du conteneur.**

Les conteneurs séquentiels principaux sont les patrons de classes **`std::vector`**, **`std::deque`** et **`std::list`**.

Exercice 1. Patron de tableaux dynamiques 1D : std::vector

L'utilisation des tableaux dynamiques standards demandera l'include <vector>.

Quelques méthodes :

- empty() permet de savoir si la taille actuelle est 0 (0 élément de présent) ;
- push_back(elm) permet d'insérer un élément à la fin (ajout en queue) ;
- pop_back permet de supprimer le dernier élément s'il existe (et dans ce cas la taille est effectivement réduite de 1) ;
- front() renvoie le premier élément ;
- back() renvoie le dernier élément ;
- size() renvoie la taille (nombre d'éléments) du conteneur ;
- resize(int n) permet de modifier la taille du conteneur ;
- capacity() renvoie la capacité du conteneur ;
- reserve(int n) permet de modifier la capacité du conteneur ;
- shrink_to_fit() permet de réduire la capacité à la taille du conteneur ;
- swap( vector ) permet d'échanger le contenu de 2 vectors **de même type** ;
- clear() permet de supprimer tous les éléments du vector ; la taille après un appel à clear() est 0, par contre il n'y a aucune garantie sur la mise à jour de la capacité (*i.e. si on clear un vector v d'un million d'entiers, la mémoire allouée peut rester inchangée* ; afin de forcer la réallocation, faire vector<int>().swap( v ) ) ;
- **Il existe un constructeur pouvant prendre 0, 1 ou 2 arguments effectifs** ; le 1er argument pour fixer le nombre d'éléments insérés (au départ) et le 2nd argument pour fixer la valeur d'initialisation des éléments insérés ; si aucune valeur d'initialisation n'est fournie alors le constructeur par défaut du type T sera appelé, sinon ça sera le constructeur par copie du type T.
- Il existe un constructeur par recopie ;
- Il existe un constructeur basé sur une paire d'itérateurs (un début et une fin) ;
- Il existe un opérateur d'affectation = (le type doit être exactement le même), un opérateur [] ;
- Il existe une méthode assign(iterDebut, iterFin) pour faire une affectation à partir d'un conteneur d'un autre type, mais contenant le même type d'élément.

**Un algorithme est un patron de fonctions dont le paramètre est le type des itérateurs fournis en arguments.** L'utilisation des algorithmes standards demandera l'include <algorithm>.

Voici quelques algorithmes pour les tableaux dynamiques :

- Il existe 2 algorithmes std::fill (ou std::fill_n) et std::generate (ou std::generate_n) pour remplir un vector (mais qui ne modifient pas la taille du vector) ;
- std::copy pour copier (sans allouer) des données ; on pourra même copier des données dans un autre type de conteneur (par exemple un std::list) ;
- std::min_element ; std::max_element ;
- std::count pour compter le nombre d'occurrences d'une valeur dans un intervalle ;
- std::for_each pour appliquer une fonction particulière à les éléments d'un conteneur ; std::transform est similaire, mais permet de choisir la destination du résultat ;
- std ::sort pour trier ;

- std::find_if pour trouver le premier élément dans un intervalle donnée qui vérifie le **prédicat unaire** fourni (*fonction avec 1 argument et qui retourne un **bool***) ; std::find.

**Travail à effectuer :**

1. Est-ce que std::vector possède un itérateur unidirectionnel, bidirectionnel ou un itérateur à accès direct ? En déduire si le conteneur std::vector possède un reverse_iterator.
2. Afficher un `vector` de 5 entiers dans le sens direct puis dans le sens inverse en utilisant des itérateurs. En déduire une fonction pour afficher dans le sens direct un vecteur de n'importe quel type muni de l'opérateur <<.
3. Afficher la taille et la capacité d'un vector. Conclure.
4. Tester tous les constructeurs de `vector`. En particulier construire un `vector` à partir d'un tableau usuel `int tab[] = {1,2,3,4,5} ;` .
5. Comment vider un `vector` d'entiers en utilisant seulement les méthodes empty() et pop_back() ? Y-a-t-il une différence par rapport à un appel direct à la méthode clear() ?
6. Parcourir et supprimer les entiers pairs d'un vector d'entiers (méthode erase(iter)).
7. Parcourir un tableau trié d'entiers et insérer à la suite d'une séquence de chiffres identiques le nombre total de ce chiffre (méthode insert(iter,val)). Par exemple 12223344444 donnera 11322354.
8. Tester les algorithmes (inclure le fichier d'en-tête algorithm) std::fill, std::fill_n, std::generate, std::generate_n et std::copy.
9. Utiliser les algorithmes std::min_element, std::max_element et std::sort pour trouver, respectivement, le min, le max et trier un tableau de flottants par ordre décroissant. Comment trier la première moitié du tableau par ordre croissant et la seconde moitié par ordre décroissant ? Comment trier un `vector` de Personne (une classe) ? Comment afficher tous les éléments d'un `vector` via l'algorithme for_each ?
10. Est-ce que les opérateurs ==, !=, <, <=, >, et >= sont définis pour les `vector` de scalaires? De string ? De type utilisateur (une classe) ?

**Exercice 2.** Patron de tableaux dynamiques 1D sans contiguïté globale des données : **std::deque**

L'utilisation des tableaux dynamiques à double fin demandera l'include <deque>.

- **Interface très similaire à celle du std::vector**, avec 2 méthodes supplémentaires notables : push_front(elm) et pop_front() ;
- Un deque ne dispose pas des méthodes capacity et reserve.

**Travail à effectuer :**

- Tester les nouvelles méthodes push_front et pop_front.

**Exercice 3.** Patron de listes doublement chaînées : **std::list**

L'utilisation des listes doublement chaînées demandera l'include <list>.

Les listes doublement chaînées (std::list) sont plus coûteuses en mémoire et en temps de calculs qu'une liste simplement chaînée si l'on doit seulement parcourir la liste « vers l'avant », i.e. dans l'ordre d'insertion des éléments.

Par contre, dès qu'il s'agit d'insérer ou de supprimer un élément à une position précise de la liste (via un iterator), l'algorithme d'insertion ou de suppression a besoin de l'iterator à la position précédente, et donc la liste simplement chaînée devient très coûteuse en temps de

calculs (linéaire en le nombre d'éléments) par rapport à la liste doublement chaînée (insertion et suppression en temps constant). **Nous n'étudierons que les listes doublement chaînées.**

**Travail à effectuer :**

1. **Tester** l'affectation =, la construction par recopie et les méthodes front, back, push_front, pop_front, push_back, pop_back, insert, erase, swap, resize, clear, remove, unique, sort, merge, reverse **sur des listes d'entiers.**
2. Comment insérer un élément en tête avec la méthode insert ? Et comment insérer en queue ? Comment insérer une `list` à la fin d'une autre list ?
3. Vous remarquerez qu'une `list` est pourvue de la méthode sort (alors que sort est utilisé comme un algorithme pour trier un tableau dynamique) ; utiliser sort pour trier une list de string non triée initialement.
4. Tester l'algorithme std::count_if (avec la condition >5) sur une list d'entiers ; utiliser l'algorithme std::copy pour copier la liste résultat dans un `vector`.
5. Parcourir et afficher une liste d'entiers via un iterator, puis via un const_iterator. Quelle est la différence entre les 2 ?
6. Parcourir la liste d'entiers via un reverse_iterator, puis via un const_reverse_iterator.
7. Déclarer une liste d'entiers, une liste de `Personne` (classe utilisateur), une liste de `Personne *` ; de quels constructeurs doit être munie la classe `Personne` pour une utilisation du constructeur par recopie de `list`, et pour une utilisation de la méthode resize ?
8. Supprimer les entiers impairs d'une liste d'entiers (via la méthode remove_if).
9. Calculer la somme des entiers pairs d'une liste d'entiers (std::accumulate).

Votre patron vous demande un programme qui gère des personnes qui entrent et sortent d'une salle de poker en ligne. N'importe quel joueur peut sortir n'importe quand et de nouveaux joueurs peuvent joindre la table s'il y a moins de 10 joueurs de présents actuellement.
En supposant que la classe `Joueur` existe, quelle est le meilleur choix de conteneur parmi les suivants (justifiez votre réponse) :

- `vector < Joueur >`
- `vector < Joueur * >`
- `list < Joueur >`
- `list < Joueur * >`

## Etude des **conteneurs associatifs**, de leurs itérateurs et des algorithmes existants sur les conteneurs associatifs

Dans un conteneur associatif, les éléments sont identifiés par une clé et ils sont ordonnés selon une relation d'ordre sur la clé. Il y a 2 grands types de conteneurs associatifs. Les ensembles (`std::set`) et les maps (`std::map`). Dans le premier, la valeur contenue et la clef sont identiques, **dans le second, on stocke des paires**, la première valeur (first) étant la clef, la seconde (second) la valeur réellement utilisée.

Les conteneurs associatifs sont pourvus d'une méthode find( cléElm ) pour trouver en temps sous-linéaire un élément (via une structure d'arbre binaire de recherche).

**Exercice 4.** Patron d'ensembles simples (on n'admet pas plus d'une copie d'un élément) : **std::set**

L'utilisation des ensembles demandera l'include <set>.

La STL fournit des algorithmes permettant de réaliser les opérations ensemblistes les plus courantes (union, intersection etc.) ; leur nom est du type set_*operation* (std::set_union, std::set_difference…)

**Travail à effectuer :**

1. Créer deux ensembles se recouvrant :
    a. l'un contenant les 10 premiers éléments de la suite de Fibonnacci ;
    b. l'autre contenant la suite des 10 premiers nombres impairs.
2. Utilisez l'opération d'union en :
    a. renvoyant le résultat à l'écran ;
    b. stockant le résultat :
        i. dans un ensemble ;
        ii. dans une deque.
3. Réalisez la même opération avec l'intersection.
4. Essayez les deux opérations de différence et de différence symétrique sur ce même exemple.

**Exercice 5.** Patron de tableaux associatifs simples (unicité des clés) : **std::map**

L'utilisation des tableaux associatifs demandera l'include <map>.

Le conteneur std::map contient des paires clé-valeur, ainsi si it est un itérateur valide sur un élément d'un std::map, (*it).first désignera la clé et (*it).second désignera la valeur associée.

**Travail à effectuer :**

1. Associer les entiers 1, 2, …, 12 aux chaînes de caractères « janvier », « fevrier », …, « decembre » ; Parcourir et afficher ce tableau associatif dans le sens croissant et dans le sens décroissant ; afficher les mois par ordre lexicographique ;
2. Tester les méthodes find, insert et erase sur un type scalaire et sur un type utilisateur.
3. Si on utilise un tableau associatif pour associer une clé à un objet, par exemple associer le nom d'une personne à son Compte (classe Compte vue dans un précédent TP), est-il plus judicieux de déclarer un map<string, Compte> ou un map<string, Compte*>. Justifier votre réponse.

# Les fichiers séquentiels (Partie non-évaluée)

Les fichiers permettent de conserver dans de la mémoire morte des données, mais aussi de transmettre ces données. Ils sont donc essentiels à la majorité des applications et systèmes informatiques.

Exercice 6. Les fichiers séquentiels en C++ : **lecture via std::ifstream** et **écriture via std::ofstream**
L'utilisation des fichiers séquentiels demandera l'include <fstream>.

Insérer des données dans un fichier :
**Pour associer un flot de sortie à un fichier** (ascii ou binaire), **il suffit de créer un objet de type ofstream.** Le 1er argument du constructeur de la classe ofstream est le nom du fichier (chaîne de caractères style C avant C++ 11 et std::string autorisé après), le second argument (optionnel) est le mode d'ouverture (par exemple ios::out | ios::binary pour signaler l'ouverture en écriture et au format binaire). Le mode d'ouverture peut également préciser si l'on doit effacer le contenu d'un fichier existant de même nom (ios::trunc) ou si l'on doit continuer à la suite du contenu existant (ios::app). Pour envoyer une information sur un flot de sortie, on utilise l'opérateur <<.

```
// ECRITURE DANS UN FICHIER toto.txt en mode texte (.txt)
ofstream f ("toto.txt"); // création d'un objet de type ofstream
if (!f.is_open())
   cout << "Impossible d'ouvrir le fichier en écriture !" << endl;
else
{
   f << 5 << " " << 8 << " " << 4.2f << endl;
   f.close(); // fermeture du flot de fichier à la fin de son
              // utilisation (si ouverture réussie)
}

// ECRITURE DANS UN FICHIER toto.raw en mode binaire (données brutes)
ofstream f ("toto.raw", ios::out | ios::binary);
if (!f.is_open())
   cout << "Impossible d'ouvrir le fichier en écriture !" << endl;
else
{
   int a=5, b=8;
   float c = 4.2f ;
   f.write( (char*)&a, sizeof(int) );
   f.write( (char*)&b, sizeof(int) );
   f.write( (char*)&c, sizeof(float) );

   f.close(); // fermeture du flot de fichier à la fin de son
              // utilisation (si ouverture réussie)
}
```

Extraire des données depuis un fichier :

**Pour associer un flot d'entrée à un fichier**, cette fois **on crée un objet de type ifstream**. Pour le mode d'ouverture, on aura besoin du mode ios::in (ouverture en lecture).

Pour obtenir une information depuis un flot d'entrée, on utilise l'opérateur >>.

```
// LECTURE DEPUIS UN FICHIER toto.txt en mode texte (.txt)
ifstream f ("toto.txt"); // création d'un objet de type ifstream
if (!f.is_open())
   cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
else
{
   int a, b ;
   float c ;
   f >> a >> b >> c ;
   f.close(); // fermeture du flot de fichier à la fin de son
              // utilisation (si ouverture réussie)
}

// LECTURE DEPUIS UN FICHIER toto.raw en mode binaire (données brutes)
ifstream f ("toto.raw", ios::in | ios::binary);
if (!f.is_open())
   cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
else
{
   int a, b ;
   float c ;
   f.read( (char*)&a, sizeof(int) );
   f.read( (char*)&b, sizeof(int) );
   f.read( (char*)&c, sizeof(float) );
   f.close(); // fermeture du flot de fichier à la fin de son
              // utilisation (si ouverture réussie)
}
```

**Chaque fois qu'un flot de fichier sera ouvert, il faudra le fermer via la méthode close() à la fin de son utilisation.**

Travail à effectuer :

1. Ecrire dans un fichier une liste de float, puis lire ce fichier afin de construire une `std::list` de float.
2. Proposer une solution pour **gérer les notes des étudiants de manière persistante** via un fichier texte et un conteneur adéquat (les lire depuis un fichier, ajouter une note à un étudiant particulier, et sauvegarder l'ensemble des notes actuelles).