E12

But : découvrir la STL (Standard Template Library).

Partie I – Conteneurs : vector

La STL dispose de nombreux types de conteneurs, définis sous forme de templates, donnant accès aux structures de données les plus courantes (tableau, liste chaînée, file, …).

Nous nous intéressons dans cette partie au container vector (include <vector>) qui se comporte comme un tableau se redimensionnant automatiquement.
Voici un exemple illustrant l'utilisation de ce conteneur avec des objets de type Date :

- Création et ajout :

```
vector<Date>  vect;     // vecteur d'objets Date, vide
Date d1(6, 10, 2021);
vect.push_back(d1);     // ajoute l'objet d1 (une copie) en fin
                        // du vecteur
```

- Parcours et accès :

```
int nb = vect.size();   // nb d'éléments du vecteur
pour i de 0 à nb
      on peut utiliser vect.at(i) qui retourne une référence sur
      le ième objet
      ou vect[i] car l'opérateur [] a été surchargé
```

1) Créer une classe Personne avec comme données (privées) le nom (string) et l'âge (int). La munir d'un constructeur, d'une fonction d'affichage et par la suite de toute fonction utile.

2) Lire un fichier texte contenant une liste de personnes avec nom et âge :
   toto 35
   untel 22
   etc

   Mettre le résultat de la lecture dans un vecteur d'objets Personne, puis l'afficher.

Partie II – Itérateurs

Un itérateur est un objet permettant de parcourir le vecteur et s'utilisant comme un pointeur sur les éléments du vecteur.

Voici un exemple illustrant l'utilisation d'un itérateur sur un vecteur d'objets Date :

```
vector<Date>  vect;
…
vector<Date>::iterator  it;
for (it = vect.begin(); it != vect.end(); it++)
      it->afficher();  // fonction afficher de la classe Date
                       // invoquée sur l'objet Date pointé par it
```

1) Afficher le vecteur d'objets Personne avec un itérateur.

Remarque : un vecteur peut être parcouru avec un indice ou avec un itérateur, d'autres types de conteneur ne peuvent être parcourus qu'avec un itérateur.

Partie III – Conteneurs : map

Le conteneur map (include <map>) est un conteneur dans lequel les éléments sont repérés par une clé. Chaque valeur de clé est unique.

Voici une illustration utilisant la classe Personne avec comme clé le nom :

- Création :

```
map<string, Personne>   map1;
// conteneur map constitué d'objets Personne avec un objet
// string comme clé
```

- Ajout d'un nouvel élément :
  L'élément ajouté est constitué d'une paire (clé, Personne).

```
pair<string, Personne>  elem(n,P);      // n est la clé (le nom),
                                        // P l'objet Personne
map1.insert(elem);
```

  Si un objet Personne de même clé est déjà présent dans la map, l'insertion retourne une erreur.

- Parcours :
  Il faut utiliser un itérateur.

```
map<string, Personne>::iterator  it;
for (it …)  {
    // l'itérateur it ne pointe pas sur un objet Personne
    // mais sur un objet pair<string, Personne>
    it->first : la clé
    it->second : l'objet Personne
}
```

1) Reprendre les questions précédentes mais avec une map.

   Remarque : la map range ses éléments par clé croissante en utilisant l'opérateur < de la classe de la clé. La classe string possède un tel opérateur comparant alphabétiquement.

2) Vérifier, en mettant des noms en doublon dans le fichier d'entrée, que la map ne contient que la première personne de chaque nom.
   Remarque : si on veut tester le retour de insert pour savoir si l'insertion a réussi :

```
pair<map<string, Personne>::iterator, bool>  ret;
ret = map1.insert(elem);
if (ret.second == false)
    // l'insertion a échoué
```

3) Chercher dans la map par la clé : saisir un nom, chercher dans la map la personne de ce nom et l'afficher.
   A utiliser : fonction find(cle) qui retourne un itérateur sur l'élément trouvé ou end() si non trouvé.

Partie IV – Algorithmes

Le but est d'appliquer sur les conteneurs des algorithmes (tri, recherche, …) fournis par la STL. Include : <algorithm>. On prend comme exemple le tri.

La STL fournit une fonction sort qui trie un conteneur par ordre croissant. On lui indique la plage du tri et la fonction de comparaison à appliquer représentée par un foncteur, un objet jouant le rôle d'une fonction.

Voici une illustration avec un vecteur d'objets Date et une plage de tri qui couvre tout le vecteur :

```
vector<Date>  vect;
…
sort(vect.begin(), vect.end(), un objet foncteur);
```

Pour le foncteur il faut créer une classe surchargeant l'opérateur de fonction () :

```
class CompareDate
{
public :
      bool operator() (const Date& d1, const Date& d2)
      {
            doit retourner true si d1 est "inférieur" à d2 selon le
            critère choisi (par ex. ordre chronologique)
      }
};
```

Ensuite on passe un objet foncteur CompareDate à la fonction sort :

```
CompareDate  cmp;
sort(debut, fin,  cmp);
```

ou en forme plus courte en passant un objet CompareDate anonyme :

```
sort(debut, fin, CompareDate());
```

1) Appliquer ce principe pour trier le vecteur de personnes par âge, puis par ordre alphabétique du nom.

Partie V – Conteneurs : list

Le conteneur vector est efficace pour l'accès à un élément et l'ajout et la suppression à la fin.
Il n'est pas performant pour un ajout ou une suppression à un emplacement quelconque.
Pour cela on utilise le conteneur list qui est une liste chainée (doublement, chainage avant et arrière). Il se parcourt avec un itérateur.

On récupère la classe Point et le fichier de points de l'exercice E9. En s'aidant de la documentation en ligne de list :

1) Lire le fichier de points et ranger le résultat dans une liste de points.

2) Afficher cette liste.

3) Supprimer dans la liste le point le plus éloigné de l'origine.

4) Choisir un point P1 présent dans la liste, créer un nouveau point P2 et l'ajouter dans la liste avant P1.