HLIN603
Feuille de TD/TP N°4 : Généricité paramétrique & STL

Exercice 1 : Modèles de fonctions et Modèles de classes

1) Proposez des modèles de fonctions ***echange*** et ***triBulles*** paramétrés par le type des éléments échangés ou triés, puis écrivez un programme qui les utilise pour trier un tableau de chaînes de caractères.

```
void echange(int& e1, int& e2)
      {int aux=e1; e1=e2; e2=aux;}

void triBulles(int T[], int tailleT)
{
     int i=tailleT-2,j; bool ech=true;
     while (i>=0 && ech)
          {
              ech=false;
              for (j=0; j<=i; j++)
              if (T[j]>T[j+1])
                    {echange(T[j],T[j+1]); ech=true;}
               i--;
          }
}
```

2) Soient les classes partielles suivantes représentant respectivement des ***œufs***, des ***bouteilles***, et des ***casiers*** de 6 bouteilles ou de 6 œufs :

```
class Bouteille { ... };
```

```
class Œuf { ... };
```

```
class CasierBouteille {
private:
     Bouteille* cases[6];
public:
     CasierBouteille();
     virtual ~CasierBouteille();
     virtual void range(Bouteille* bouteille, int numeroCase);
...};

CasierBouteille::CasierBouteille()
      {for (int i=0; i<6; i++) cases[i]=NULL;}

CasierBouteille::~CasierBouteille(){}

void CasierBouteille::range(Bouteille* bouteille, int numeroCase)
      {cases[numeroCase]=bouteille;}
...
```

```
class CasierOeuf {
private:
     Oeuf* cases[6];
public:
     CasierOeuf();
     virtual ~CasierOeuf();
virtual void range(Oeuf* oeuf, int numeroCase);
};
```

```
CasierOeuf::CasierOeuf()
    {for (int i=0; i<6; i++) cases[i]=NULL;}

CasierOeuf::~CasierOeuf(){}

void CasierOeuf::range(Oeuf* oeuf, int numeroCase)
    {cases[numeroCase]=oeuf;}
```

2.a) Proposez, en donnant le fichier .h et le fichier .cc, un modèle de classe **Casier6** paramétré par un type de produit et qui généralise les casiers de bouteilles et casiers d'œufs.

2.b) Ecrivez ce qu'il faut pour :
- Instancier le modèle Casier6 pour obtenir un casier d'œufs.
- Créer un œuf (on suppose que la classe dispose d'un constructeur sans paramètres).
- Le ranger dans la case 4 du casier.

Exercice 2 : Une classe paramétrée Dictionnaire

Un dictionnaire est un ensemble d'associations, c'est-à-dire de couples (clé, valeur), où la clé est d'un type donné ***TypeCle*** et la valeur d'un type donné ***TypeValeur***. Une association peut être représentée par la classe ***Assoc*** vue en cours. Un dictionnaire possède habituellement les méthodes suivantes :
- put: place une clé et une valeur associée dans le dictionnaire;
- get: prend une clé et renvoie la valeur associée;
- estVide: dit si le dictionnaire est vide;
- taille: retourne le nombre d'associations clé-valeur effectivement présentes dans le dictionnaire;
- contient: prend une clé et dit si elle est présente dans le dictionnaire;
- affiche: affiche le contenu du dictionnaire sur un flot de sortie.

1) **Spécification.** Écrivez la partie "signatures des méthodes" du fichier .h correspondant au dictionnaire générique.
   - Ajoutez la surcharge de ***l'opérateur <<*** pour l'affichage et la surcharge de ***l'opérateur =*** pour l'affectation.
2) **Implémentation par un tableau d'associations.** Le tableau d'associations est alloué dynamiquement, par défaut avec une taille de 10. L'indice d'une association est calculé grâce à une fonction de hachage appliquée à la clé (voir en annexe).
   - En utilisant les exemples de résultats de la fonction de hachage, faites quelques essais d'insertions "à la main" pour voir comment les choses se passent : par exemple put("abricot",235); put("amande",1023); put("ananas",242); put ("pomme",83); ...
   - Complétez la partie "attributs" du fichier .h, et écrivez le fichier .cc correspondant.

Pour faciliter l'écriture de certaines méthodes, on peut écrire la méthode privée :

```
void CherchCl(const TypeCle& cl, int& i, int& res);
/* cherche la cle cl dans le dictionnaire :
      si cl est presente: renvoie res=1, i indice de la case de cl dans T;
      si cl est absente et le dictionnaire non plein: renvoie res=0, i indice de case possible pour cl dans T;
      si cl est absente et le dictionnaire plein: renvoie res=2, i non significatif. */
```

3) **Utilisation.** Instanciez vos classes pour avoir un dictionnaire dont les clés sont des chaînes et les valeurs des entiers.
   - Écrivez un programme simple qui teste le fonctionnement de ce dictionnaire (ajoutez des couples, affichez le dictionnaire, rajoutez des couples pour tester l'agrandissement, affichez, etc.).
   - Utiliser la classe dictionnaire pour stocker les mots d'un texte lu sur l'entrée standard et comptabiliser le nombre d'occurrences de chacun.

Exercice 3 : STL (Standard Template Library)

Compléments sur les conteneurs. L'interface des classes conteneurs contient un ensemble de définitions de types publiques, telles que (pour un conteneur paramétré par le type T) :

- **typedef T value_type**, type des éléments,
- **typedef ... iterator**, type des itérateurs sur le conteneur (la partie en pointillé dépend de l'implémentation choisie),
- **typedef ... const_iterator**, type des itérateurs constants sur le conteneur (la partie en pointillé dépend de l'implémentation choisie).

Ces définitions de types permettent d'écrire des fonctions très générales, s'appliquant à tous les conteneurs. Pour les utiliser, il faut les faire précéder du mot-clef ***typename***. Le code suivant définit fonction somme que nous vous invitons à examiner pour faire la somme des éléments d'un vecteur d'entiers et d'un set de float.

```
template<typename C>
// le parametre doit être un conteneur
// C::value_type doit etre un type muni de l'addition
// et 0 son élément neutre
typename C::value_type somme(const C&c)
{
typename C::value_type s=0;
typename C::const_iterator p = c.begin();
while (p != c.end()){s+=*p; p++;}
return s;
}
```

- **Comptage de mots.** Utilisez la classe ***map***, pour compter le nombre de mots d'un texte et afficher le résultat du comptage.
- **Effacement conditionnel**. Nous vous proposons d'examiner l'algorithme d'effacement conditionnel ***remove_if***. Sa sémantique est très particulière : la fonction ne supprime pas d'éléments en mémoire, elle les écrase, et retourne la fin valide du conteneur. Remarquez au passage que les tableaux se manipulent comme les conteneurs, et où se trouve l'itérateur (la position) de fin du conteneur ! Essayez les mêmes manipulations avec un vecteur d'entiers. Notez le type de la fonction passée en paramètre à ***remove_if***.

```
bool odd (int a)
{ return a % 2;}
int numbers[6] = { 0, 0, 1, 1, 2, 2 };

int main ()
{
int* ite =
remove_if (numbers, numbers + 6, odd);
for (int i = 0; i < 6; i++)
cout << "adresse element " << i << " = " << (int)(&numbers[i])
<< " valeur = " << numbers[i] << endl;
cout << endl;
cout << "adresse retournee par remove_if =" << (int)ite;
cout << endl;
return 0;
}
```

- **Un effacement conditionnel plus satisfaisant**. Ecrivez une fonction ***MonRemove_if***, paramétrée par un conteneur et un prédicat (une fonction booléenne) qui supprime effectivement les éléments du conteneur vérifiant le prédicat. Utilisez ***remove_if*** et ***erase*** dans un programme de test.
- **Classes fonctions**. Vous trouverez ci-dessous une classe dont les instances peuvent être considérées comme des fonctions testant la divisibilité.

```
class divisiblePar {
private:
int diviseur; // donnée membre : le diviseur
public:
divisiblePar(int div) : diviseur(div) {}; // constructeur
bool operator()(const int& dividende) { //
return (dividende%diviseur)==0 ;
}
};
```

Analysez-la. Comparez cette classe et ses instances à une fonction classique ***divisiblePar*** à deux paramètres.

```
void TesteDivisiblePar()
{
divisiblePar f(2);
divisiblePar g(3);
cout << "f(4) = " << f(4) << endl;
cout << "f(8) = " << f(8) << endl;
cout << "f(5) = " << f(5) << endl;
}
```

Que vaudrait g(9) ?

- **Affichage.** Ecrivez un modèle de fonction d'affichage, paramétrée par un type C, supposé être un type de conteneur, prenant comme paramètre un conteneur (passé par référence), et affichant sur le flot de sortie standard cout tous les éléments du conteneur. Vous utiliserez ***C:: value***, qui représente le type des éléments du conteneur, pour paramétrer l' ***ostream_iterator***.

- **Crible d'Erathostènes** On souhaite connaître (en les affichant, par exemple) tous les nombres premiers jusqu'à une certaine borne B. Utilisez une liste (list en STL) pour stocker tous les entiers de l'intervalle [2, B]. Puis supprimez successivement tous les multiples des nombres premiers rencontrés, en vous servant de ***MonRemove_if***. Vous pouvez extraire ou non les nombres premiers de la liste quand vous les trouvez.
- **Ensemble ordonné.** Cet exercice est volontairement plus libre et vous demande un peu de conception préalable. Ecrivez la classe ***EnsembleOrdonne*** vue en cours. Les paramètres de généricité sont le type des éléments stockés et une fonction de comparaison de ces éléments. On doit pouvoir ajouter des éléments, en retirer, retourner les plus petits ou les plus grands éléments, etc.

Annexe : Fonction de hachage (Pour Exercice 2)

Voici un exemple de fonction de hachage simple :

```
#include<string>
int hash(string s, int tailleTab)
{int i=0;
//calcul d'un entier associe a la string
for (int j=0; j<s.length(); j++) i=i+(j+1)*s[j];
//adaptation a la taille du tableau
return (i % tailleTab);
}
```

Voici quelques résultats produits par cette fonction :

taille tableau = 10	taille tableau = 15
hash(abricot, 10) = 8	hash(abricot, 15) = 13
hash(amande, 10) = 2	hash(amande, 15) = 7
hash(pomme, 10) = 2	hash(pomme, 15) = 12
hash(ananas, 10) = 3	hash(ananas, 15) = 3
hash(prune, 10) = 6	hash(prune, 15) = 1
hash(griotte, 10) = 3	hash(griotte, 15) = 13
hash(poire, 10) = 0	hash(poire, 15) = 5
hash(orange, 10) = 1	hash(orange, 15) = 1
hash(citron, 10) = 8	hash(citron, 15) = 3
hash(mangue, 10) = 6	hash(mangue, 15) = 1
	hash(papaye, 15) = 6

Lorsque la fonction hash produit une même valeur pour deux clés différentes, on est dans une situation de collision. La méthode de résolution de collisions la plus simple à implémenter est la méthode “recherche séquentielle simple” : on recherche, à partir de la case qui aurait dû être la bonne, la première case libre (si on arrive en fin de tableau, on recommence au début), et on place l'association dedans. Cette méthode n'est pas excellente (elle crée des “grappes”), mais elle suffit pour tester cet exercice.