Responsable pédagogique	AF	**AM**	OP	PM
Période	Sem1	Sem2	**Sem3**	Sem4
Volume horaire	Cours/TD		TP	
			6	

[CPP]
ACRONYMES & STL

Indicateur temporel (hors rédaction du compte-rendu) :

questions	1h	2h	3h	4h	5h	6h
1 .. 4						
5 .. 7						
8 .. 13						
14 .. 15						
16 .. 17						
18 .. 19						

Documents à rendre : Rapport de TP complet (modèle LLF IRIS)
Objectifs : Mise en œuvre de quelques modèles de la STL
Ressources : Base de données « acronym4.dta »

Cahier des Charges

acronyme (n.m.) mot constitué par les premières lettres des mots qui composent une expression complexe ; comme par ex. COBOL : COmmon Business Oriented Language (*Larousse*)

Les acronymes sont pour le moins très utilisés de nos jours (notamment dans le domaine informatique). On se propose ici d'employer ces jolies petites choses comme support à l'écriture de classes au sens POO (encore un !) du terme.

Bien évidemment, histoire de compliquer le tout, un acronyme peut avoir plusieurs significations sans rapport les unes avec les autres, même en restant dans le même domaine (technologique ou non...). Prenons par exemple PM qui pour nous peut signifier *Physical Medium* mais qui pour d'autres fait plutôt penser à Police Militaire...ou à Pistolet Mitrailleur...

De même, il n'est pas de bon goût de confondre un Centre de Documentation et d'Information avec un Contrat à Durée Indéterminée...ou le Journal Officiel avec les Jeux Olympiques !

Un objet "acronyme" sera constitué pour nous de l'acronyme lui-même et de ses définitions (nombre non limité...) ; chaque signification est associée avec sa langue et pourrait se voir attribuer un domaine d'utilisation (informatique, réseau, social, militaire, sports & loisirs,...) mais ceci est HS(*) pour le moment.

Un ensemble d'objets "acronyme" sera ordonné dans un dictionnaire, interface entre l'utilisateur et l'archive sur disque des informations.

(*) comprendre au choix Hors Sujet, ou nécessitant des Heures Supplémentaires !

Modélisation d'une définition d'acronyme

Pour notre étude, nous allons simplement associer une définition de type chaîne de caractères avec un numéro traduisant la langue de cette définition.

SNIR-TP/ACRONYMS-v4 © Alain Menu 2021

1. Créez un nouveau répertoire de travail et y placer l'embryon de classe suivant (fichier **cdef.h**) :

```
#ifndef CDEF_H
#define CDEF_H

#include <string>
using namespace std ;

class CDef {

      static string     cl[] ;

  public :
      CDef() {}
      CDef(const string& definition, int langue = 0 ) {}
      CDef(const CDef& cdef ) {}
      ~CDef() {}

      static string     CodLang(int langue ) {
            int i = 0 ;
            while ( ! cl[i].empty() ) {
                  if ( i == langue )      return cl[i] ;
                  i++ ;
            }
            return cl[i] ;
      }
} ;

//                      0     1     2     3     4     5     6
string CDef::cl[] = { "EN", "FR", "DE", "SP", "IT", "JP", "RU", "" } ;

#endif
```

Ce code contient une méthode statique capable de renvoyer l'abréviation d'une langue à partir d'un numéro. La correspondance est maintenue par le tableau **cl** qui pourra être complété si besoin (cf. http://www.abbreviations.com/acronyms/LANGUAGES2L).
La langue par défaut est l'anglais (EN).

2. Ajoutez des membres privés (nommés par exemple **def** et **lang**) et leurs accesseurs *get/set* publics susceptibles de maintenir le texte de la définition et son code de langue.

3. Complétez les 3 constructeurs (et si nécessaire le destructeur) en conséquence.

4. Proposez un programme minimaliste **testdef** permettant de valider la classe **CDef** (création d'objets, utilisation de l'interface de la classe,...).

Modélisation d'un acronyme et de ses définitions

Il nous faut maintenant créer une classe susceptible d'associer un acronyme (de type **string**) et un ensemble d'objets **CDef** pour la liste de ses significations.

La classe abstraite **CAcroBase** ci-dessous propose un modèle de développement :

```
#ifndef CACROBASE_H
#define CACROBASE_H
```

```
#include "cdef.h"
#include <string>
using namespace std ;

class CAcroBase {

  public :
      virtual string    acro() const = 0 ;
      virtual int       size() const = 0 ;
      virtual CDef&     def(int index ) = 0 ;
      virtual void      addDef(const CDef& def ) = 0 ;
} ;

#endif
```

Les 4 méthodes virtuelles pures proposent respectivement :

- la récupération de l'acronyme,
- le nombre de définitions enregistrées,
- l'accès à une des définitions par son numéro d'ordre,
- la possibilité d'ajouter une nouvelle définition.

5. Créer une nouvelle classe **`CAcro`** dérivée de **`CAcroBase`** (donc dans deux fichiers séparés **`cacrobase.h`** et **`cacro.h`**).
   Ajouter à la classe **`CAcro`** les éléments suivants :
   - un attribut pour stocker l'acronyme,
   - un constructeur par défaut qui fixe l'acronyme à `"?"`,
   - un constructeur acceptant l'acronyme en argument,
   - un constructeur de copie,
   - les 4 méthodes virtuelles du modèle (avec les 3 dernières vides pour le moment).

6. Enrichir la classe avec un membre privé **`list<CDef>`**. Ce conteneur STL est bien évidemment destiné au stockage des définitions.
   Finaliser le code de la classe en conséquence :
   - constructeur de copie,
   - méthode **`size()`** : voir méthode homonyme du conteneur **`list`**,
   - méthode **`addDef()`** : vous pouvez utiliser **`push_back()`** pour l'ajout,
   - méthode **`def()`** : utilisez un itérateur !

7. Proposez un programme minimaliste **`testacro`** permettant de valider la classe **`CAcro`** (création d'objets, utilisation de l'interface de la classe,...).

Modélisation d'un dictionnaire d'acronymes

On considère qu'un dictionnaire d'acronymes est matérialisé par un fichier texte tel que celui fourni gracieusement en support (fichier **`acronym4.dta`**).

8. Éditez le fichier et observez son contenu. vous noterez qu'il n'y a qu'une et une seule information par ligne, et que des commentaires ou des lignes vides peuvent apparaître. Comment est indiqué un acronyme ? comment sont signalées ses définitions ?

Là encore nous allons utiliser une classe abstraite comme modèle de développement. Le code ci-dessous montre une solution qui va prendre en charge le processus de chargement du contenu d'un fichier « dico. d'acronymes » :

```
#ifndef CDICOBASE_H
#define CDICOBASE_H

#include "cdef.h"
#include <iostream>
#include <fstream>
using namespace std ;

class CAcro ;

class CDicoBase {
      string      path ;
      fstream     stream ;
 public :
      CDicoBase(const string& path = 0 ) {
            this->path = path ;
            stream.open(this->path.c_str(), ios::in ) ;
      }
      virtual ~CDicoBase() {
            if ( stream )     stream.close() ;
      }

      bool isOpen() const { return stream.is_open() ; }

      bool load() {
            if ( !isOpen() )  return false ;
            // todo
            return true ;
      }

      virtual int       size() const = 0 ;
      virtual CAcro&    acro(int index ) = 0 ;
      virtual void      addDef(string acro, const CDef& cdef ) = 0 ;
      virtual void      sort() = 0 ;
} ;

#endif
```

9. Étudiez le code proposé ci-dessus.
   Pourquoi le *header* de la classe **`CAcro`** n'est-il pas inclus (ce n'est pas un oubli !) ?
   Que représente l'attribut **`stream`** ?
   Quelle information devra recevoir le constructeur ? Qu'en fait-il ?
   Quel est, à votre avis, le rôle de chacune des méthodes imposées par ce modèle ?

10. Étudiez maintenant le corps de la méthode **`load()`** ci-dessous. Rédigez une explication détaillée de son fonctionnement.

```
bool CDicoBase::load()
{
      if ( !isOpen() )  return false ;

      string      line , acro ;

      while ( getline(stream, line ) ) {

            size_t pos = line.rfind('\r') ;
            if ( pos != string::npos )    line.replace(pos,1, "\0" ) ;

            if ( line[0] == '*' ) {
```

```
                        line.erase(0, 1 ) ;
                        acro = line ;
                    }
                    if ( line[0] == '+' ) {
                        int cod = line[1] - '0' ;
                        line.erase(0, 3 ) ;
                        addDef(acro, CDef(line, cod ) ) ;
                    }
                }
                return true ;
            }
```

11. Créez une nouvelle classe **CDico** dérivée de **CDicoBase**.

    Le constructeur doit se contenter d'appeler celui de sa classe de base, ils ont la même liste d'arguments).

    Ajouter un attribut privé **acronyms** de type **vector<CAcro>**. pour le stockage des éléments **CAcro** du dictionnaire.

    Renseignez les méthodes **size()** et **acro()** en vous inspirant du travail fait lors de l'écriture de la classe **CAcro**. Les 2 autres méthodes du modèles sont laissées vides pour le moment…

12. Testez l'ensemble en utilisant cette fois le programme de test fourni en annexe 1. ni la compilation ni l'exécution ne doivent générer d'erreur, mais le résultat n'est pas encore très convaincant…

    Pourquoi ? parce qu'il faut renseigner la méthode **CDico::adddef()** !

13. Commencez par y placer les 2 instructions suivantes. Testez. Constatez. Expliquez…

    ```
    acronyms.push_back( CAcro(acro) ) ;
    acronyms.back().addDef( cdef ) ;
    ```

14. Vous l'aurez compris, si un acronyme existe déjà dans le vecteur, il ne faut pas le créer à nouveau mais simplement lui ajouter une définition. Ajoutez le code nécessaire à cette recherche préalable.

    Bon ok, un petit coup de pouce : vous aurez certainement besoin d'utiliser un itérateur, l'algorithme **find_if()** et un prédicat de comparaison de chaînes…

15. Que manque-t-il pour finir ? La méthode **CDico::sort()**. Implémentez cette méthode en utilisant l'algorithme du même nom une classe *functor* du genre « plus petit que » (vous comparez des objets **CAcro**)…

CAcro/CDico 2, le retour

À ce stade, vous disposez normalement d'une architecture d'application qui injecte sur la sortie standard le contenu d'un dictionnaire d'acronymes triés par ordre alphabétique.

Bien évidemment, l'IHM laisse à désirer… Vous avez toute liberté (en tout cas quand vous aurez fini ce TP) d'améliorer le produit :

- possibilité d'ajout d'acronymes et/ou de définitions,
- possibilité de modifications,
- sauvegarde du dictionnaire,
- … …

Mais l'objet du TP est avant tout la mise en œuvre de la STL !

Nous allons donc maintenant mieux comprendre pourquoi être passé par des modèles de développement… Sans changer un iota de la classe **CDef** ni du programme utilisateur, nous allons complètement réorganiser le fonctionnement interne des objets **CAcro** et **CDico** (« *si çà c'est pas de l'abstraction de données…* ») ; et pour la pure beauté du geste, nous allons utiliser des conteneurs associatifs !

Note : Pour ne pas tout modifier, les nouvelles versions des classes **`CAcro`** et **`CDico`** seront définies respectivement dans les fichiers **`cacromap.h`** et **`cdicomap.h`**... Seule la ligne d'inclusion dans le programme principal sera à adapter (même la ligne de commande ne change pas si toutes les méthodes sont *inline* !).

Le questionnement va être rapide (RTFM) :

16. Copier l'ancienne classe **`CAcro`** et remplacer le type de l'élément de stockage des définitions par **`multimap<int,string>`**, où la clé est le code de langue et la valeur le texte de la définition.

17. Adapter le corps des méthodes **`addDef()`** et **`def()`** en conséquence. Testez et validez.

18. Même démarche pour **`CDico`**, mais cette fois remplacez le type vecteur par **`map<string,CAcro>`**. La clé est l'acronyme, même si celui-ci est aussi présent dans la valeur...

19. Adapter en conséquence le corps des méthodes **`addDef()`** et **`acro()`** ; et ne vous plaignez pas, car pour **`sort()`**, il suffit de vider son contenu (justifiez...).

Ce petit aperçu de la puissance des conteneurs de la STL se termine ici...
Imaginez un instant (pour ceux qui ne l'ont jamais fait ; mais rassurez-vous, on survit !) devoir écrire la même application « à la main », comme on doit le faire C lorsqu'il est indispensable, et néanmoins intelligent, de mettre en œuvre des structures dynamiques...

Annexe 1 : Programme principal

```
#include "cdico.h"

#define DEFAULTDTA "acronym4.dta"

int main(int argc, char** argv )
{
      string nf = DEFAULTDTA ;
      if ( argc > 1 )   nf = string( argv[1] ) ;

      CDico dico( nf ) ;

      if ( !dico.load() )     return -1 ;
      dico.sort() ;
      cout << dico.size() << " acronymes" << endl ;

      for ( int i = 0 ; i < dico.size() ; i++ ) {
            CAcro a = dico.acro(i) ;
            cout << a.acro() << endl ;
            for ( int j = 0 ; j < a.size() ; j++ ) {
                  cout << "- " << a.def(j).def() ;
                  cout << " [" ;
                  cout << CDef::CodLang( a.def(j).lang() ) ;
                  cout << ']' << endl ;
            }
      }
      return 0 ;
}
```

Annexe 2 : Exemple de diagramme de classes UML

version « *sequence containers* »