Modelisation Objet -- MOBJ

Contents

Soumission des Comptes Rendus de TME

Le compte rendu d'un TME doit être rendu **avant** le mercredi midi suivant.
La lisibilité du code est notée.
Le compte rendu devra uniquement contenir:

- Le répertoire des sources `<top>/src/` (cf. Organisation du Code et Compilation).
- Une explication concise sous forme d'un fichier texte `CR.txt` ou PDF `CR.pdf`.

Tous ces fichiers devront être fournis sous forme d'une archive compressée de type `.tar.gz`:

```
etudiant@pc:dir> cd <top>
etudiant@pc:top> tar zcvf TME3.tar.gz src
src/
src/cBox.h
src/cBox.c
src/cMain.c
src/CMakeLists.txt
src/CR.txt
etudiant@pc:top>
```

Cette archive doit être jointe en attachement d'un e-mail envoyé à l'encadrant de votre groupe de TME, <Jean-Paul.Chaput@lip6.fr> ou <Marie-Minerve.Louerat@lip6.fr>. Le message de l'e-mail doit faire apparaître *clairement* les noms et prénoms de toutes les personnes ayant participé au dit TME.

Note

Nous n'accepterons pas de liens vers des sites tiers (stockage de type *cloud* du style de Dropbox ou Google Drive).

Note

Nous n'accepterons plus de compte rendu sous une autre forme.

TME3 -- Conteneurs & Itérateurs

Au cours de ce TME nous allons passer en revue trois principaux types de conteneurs de la STL, `vector<>`, `list<>` et `map<>`, ainsi que leurs itérateurs. Nous verrons aussi leurs relations avec les fonctions de tri.

Note

Toutes les fonctions membres des conteneurs n'ont pas été présentées en cours. Pour avoir la spécification complète, se référer à `http://en.cppreference.com/`.

Texte de Travail

Pour pouvoir travailler, nous allons utiliser les mots d'un petit texte fourni dans le fichier GPL_2_text.h. Le texte vous est fourni sous forme d'un tableau de `char*`. La fin du tableau est indiquée par un pointeur `NULL`.

```
const char* GPL_2_text[] = {
  "GNU", "GENERAL", "PUBLIC", "LICENSE",
  // ...
  NULL
  };
```

La variable `GPL_2_text` va donc être du type `char**`. En effet nous avons un *tableau* (c'est à dire un pointeur) sur *des* chaînes de caractères (c'est à dire des tableaux de `char`). Donc au final, par rapport au type `char` (un cacactère dans l'une des chaînes de caractères du tableau) nous avons bien deux *pointeurs* (soit le `char**`).

Note

Conversion entre un const char* et une string: la classe `string` possède un constructeur ayant pour argument `const char*`, ce qui implique que l'on peut initialiser (ou affecter) un objet de cette classe directement à partir d'un `const char*`. Par exemple:

```
void f ( string s ) {
  cout << "Chaine:" << s << endl;
}

int main ( int argc, char* argv )
{
  const char* texte = "Du baratin";
  string s1  (texte);
  string s2 = texte;

  f( texte ); // Conversion de "const char*" en "string".

  return 0;
}
```

Classe Chronomètre (Timer)

Afin de pouvoir mesurer le temps écoulé durant l'exécution du programme une classe `Timer` vous est fournie. Elle est à copier dans un fichier d'en tête `Timer.h`. Comme elle ne comporte que des méthodes **`inline`**, le fichier d'en tête seul suffit.

```
#include  <ctime>
#include  <iostream>

#ifndef  TIMER_H
#define  TIMER_H

class Timer {
  public:
    inline               Timer       ();
    inline Timer&        start       ();
    inline Timer&        stop        ();
    friend std::ostream& operator<< ( std::ostream&, const Timer& );
  private:
    clock_t  start_;
    clock_t  stop_;
};

inline        Timer::Timer () : start_(clock()), stop_(start_) { }
inline Timer& Timer::start () { start_ = clock(); return *this; }
inline Timer& Timer::stop  () { stop_  = clock(); return *this; }

inline std::ostream& operator<< ( std::ostream& o, const Timer& timer )
{
  clock_t delta = (timer.stop_ - timer.start_) / (CLOCKS_PER_SEC/1000);
  o << (delta/1000) << "." << (delta%1000) << " secondes ecoulees";
  return o;
}

#endif  // TIMER_H
```

Configuration de cmake

Le contenu fichier `CMakeLists.txt` pour ce TME vous est fourni:

```
# -*- explicit-buffer-name: "CMakeLists.txt<M1-MOBJ/tme3>" -*-
#
# Pour voir les commandes lancées par cmake/make, utiliser:
#     > cmake -DCMAKE_VERBOSE_MAKEFILE:STRING=YES ../src

 cmake_minimum_required(VERSION 2.8.0)
 project(TME3)

 set (CMAKE_CXX_FLAGS              "-Wall -g" CACHE STRING
                                   "C++ Compiler Release options." FORCE)
 set (CMAKE_INSTALL_PREFIX         "../install" )

    include_directories( ${TME3_SOURCE_DIR})

                     set( includes     Timer.h )
                     set( cpps         Main.cpp )
           add_executable( tme3        ${cpps} )

                  install( TARGETS tme3        DESTINATION bin )
                  install( FILES   ${includes} DESTINATION include )
```

Note

Rappel la méthode pour organiser votre code et compiler est décrite dans procédure de compilation.

Programme de Test

Petit programme de test:

```
using namespace vector_bench;

int main ( int argc, char* argv[] )
{
  backInsert     ();
  frontInsert    ();
  sortEachInsert();

  return 0;
}
```

Pour chaque conteneur, on écrira les fonctions `backInsert()`, `frontInsert()` et `sortEachInsert()` en les mettant dans des **namespace** séparés pour éviter les collisions. On les créera vides pour les remplir au fur et à mesure du TME.

Le conteneur `vector<>`

On implémentera les différentes fonctions de test du vecteur au sein d'un **namespace** `vector_bench`.

La fonction `std::sort()` est un *template* fourni par la STL via l'en-tête `<algorithm>` (elle ne fait pas partie de la classe `vector<>`).

Question 1

Écrire une fonction `backInsert()` effectuant les tâches suivantes:

- Charger dans un vecteur de `string` le texte en insérant les nouveaux éléments à la fin.
- Afficher le nombre d'éléments du vecteur.
- Trier les éléments du vecteur.
- Afficher tous les éléments du vecteur. On les affichera sur une seule ligne (ce sera très long).

Compiler et exécuter ce programme. Mesurer le temps d'exécution grâce à la classe `Timer` fournie.

Note

Le temps peut varier légèrement d'une exécution à l'autre en fonction de la charge de la machine. Lancez votre programme plusieurs fois pour avoir un temps moyen.

Question 2

Écrire une fonction `frontInsert()` identique à la précédente, mais qui, *au lieu d'insérer les éléments en fin de conteneur*, les insère en tête. Sachant que `vector<>` n'a pas de `push_front()`, comment peut-on faire (simplement).

Mesurer le temps. Que peut-on en conclure ?

Question 3

Écrire une fonction `sortEachInsert()`, qui effectue les mêmes traitements que `backInsert()` *à ceci près* que le tri, au lieu d'être effectué une seule fois en fin de fonction sera fait après l'insertion de chaque élément.

Mesurer le temps. Conclusion?

Le conteneur `list<>`

On implémentera les différentes fonctions de test des listes au sein d'un **namespace** `list_bench`.

La classe `list<>` dispose directement d'une méthode `std::sort()`, plus optimisée que celle fournie par `<algorithm>` (elle fait partie de la classe `list<>`).

Question 4

On reprend les fonctions de test des `vector<>` et on les adapte pour la `list<>`.

Mesurer les temps. Effectuer une comparaison entre les différents tests de `list<>` et une comparaison entre les tests identiques pour `list<>` et `vector<>`. Conclusions ?

Le conteneur `map<>`

On implémentera la fonction de test de la `map<>` au sein d'un **`namespace`** `map_bench`.

Question 5

Nous allons utiliser la `map<>` pour compter le nombre de fois qu'un mot apparaît dans le texte. La `map<>` aura donc pour *clé* un mot (i.e. une `string`) et pour *valeur* un compteur (i.e. un `int`).
La fonction effectuera les traitements suivants:

- Pour chaque mot de texte:
  - Chercher si un élément ayant pour clé le mot existe.
  - S'il existe, incrémenter le compteur associé.
  - S'il n'existe pas, insérer un nouvel élément dans la `map<>` (une `pair<>` (*clé*, *valeur*)) avec le compteur à 1.
- Afficher le nombre d'éléments dans la `map<>`.
- Afficher l'ensemble des éléments de la `map<>` sous forme d'une seule grande ligne de *mot:compte* (le mot, suivi du nombre de fois où il apparaît dans le texte).
- Recalculer le nombre de mots du texte.

Fonction de Tri, Objet Fonctionnel

Question 6

Créer un objet fonctionnel `CompareString` effectuant la comparaison lexicographique *inverse* sur deux `string`. Utiliser cet objet fonctionnel en tant que troisième paramètre de la `map<>` et observer le résultat à l'exécution.

Note

Objet Fonctionnel: c'est une classe, pour laquelle on a défini l'opérateur membre d'appel de fonction:

```
rtype  operator() ( atype1 arg1, atype2 arg2, ... );
```

Et qui permet d'appeler un objet de cette classe *comme si* c'était une fonction retournant `rtype` et ayant pour arguments `arg1` et `arg2`.