LO21 - Médian 2011 - Durée : 2h - Poly de cours et notes manuscrites autorisées
Chaque exercice doit être rédigé sur une copie séparée

Exercice 1 - Distributeurs.

Dans cet exercice un `distributeur` est un objet qui fournit à ses clients un mécanisme pour stocker, récupérer et éliminer des valeurs de type **`int`**. La particularité d'un distributeur est qu'il ne fournit pas de moyen pour choisir les éléments stockés qui seront récupérés ou éliminés. On veut alors développer **plusieurs genres de distributeur**, chacun se caractérisant par le choix de l'élément qu'il permet de récupérer ou d'éliminer :

- une **pile** est un distributeur qui permet de récupérer la valeur du **dernier élément stocké** ou de l'éliminer.
- une **file** est un distributeur qui permet de récupérer la valeur du **premier élément stocké** ou de l'éliminer.
- une **file_de_priorite** est un distributeur qui permet de récupérer ou d'éliminer **l'élément de plus grande valeur**.

Les caractéristiques d'une classe distributeur seront les suivantes :

- Elle devra disposer au moins des 4 méthodes suivantes. La méthode **`put`**, qui appliquée à un objet `distributeur`, permettra de stocker une valeur de type `int` passée en argument de la méthode. La méthode **`remove`** (sans argument), qui appliquée à un objet `distributeur`, permettra d'éliminer une des valeurs stockées (choisie selon les caractéristiques du distributeur). La méthode **`item`** (sans argument), qui appliquée à un objet `distributeur`, renvoie une des valeurs stockées (choisie selon les caractéristiques du distributeur) sans pour autant pouvoir la modifier. La méthode **`empty`** (sans argument) , qui appliquée à un objet `distributeur`, renvoie un booléen égal à vrai si le distributeur ne stocke aucun élément, et faux sinon.
- Il devra être possible de dupliquer (par affectation ou par construction par recopie) un objet distributeur.

Il a été décidé que la classe `file_de_priorite` serait entièrement développée par nos soins. Pour concevoir les classes `pile` et `file`, il a aussi été décidé que l'on réutiliserait au mieux une classe **`liste`** développée précédemment dont l'interface apparait dans le fichier "`liste.h`" sous cette forme :

```
class liste {
public:
	liste(); // construit une nouvelle liste
	void ajouter_en_tete(int e); // ajoute l'élément e en tête de liste
	void ajouter_en_queue(int e); // ajoute l'élément e en queue de liste
	int tete() const; // renvoie la valeur de l'élément en tête de liste
	int queue() const; // renvoie la valeur de l'élément en queue de liste
	void eliminer_tete(); // élimine l'élément en tête de liste
	void eliminer_queue(); // élimine l'élément en queue de liste
	unsigned int taille() const; // renvoie le nombre d'éléments stockés dans la liste
};
```

Un objet de la classe `liste` permet de gérer une liste de valeurs de type `int`. On peut ajouter ou éliminer des éléments en début de liste ou en fin de liste. On peut connaître la valeurs des éléments situés en début ou en fin de liste. Enfin, on peut connaître le nombre d'éléments stockés dans la liste. La classe `liste` a été implémentée de façon à ce qu'une instance de cette classe puisse être dupliquée, soit par construction par recopie, soit par affectation.

Dans quelques unes des questions suivantes, vous devrez fournir des implémentations de classes. Pour ces questions, écrivez les fichiers "**`distributeur.h`**" *et* "**`distributeur.cpp`**" *qui contiennent vos réponses. Ajouter des commentaires indiquant les questions auxquelles vous répondez. Pour les autres questions, répondez avant ou après ces deux fichiers.*

Question 1 : Exprimer sommairement (en quelques lignes) les avantages de réutiliser du code existant.

Question 2 : Expliquer (sans le faire) 2 moyens différents pour réutiliser la classe `liste` dans l'implémentation des classes `pile` et `file`.

Question 3 : Que signifie le mot clé `const` dans le prototype de la méthode `liste::tete` ?

Question 4 : Comment faire (à partir de l'interface donnée) pour changer un élément situé en tête d'un objet `liste` ? Ecrire le code d'une fonction qui permet de faire une telle opération et qui prend en paramètre un argument *représentant* un objet de la classe `liste` et une valeur de type `int` représentant la valeur qui doit remplacer la valeur de l'élément en tête de `liste`.

Question 5 : On décide d'utiliser l'héritage pour réutiliser la classe `liste` dans l'implémentation des classes `pile` et `file`. Dessiner un diagramme UML explicitant la hiérarchie de classes qui lie les classes `distributeur`, `pile`, `file`, `file_de_priorite` et `liste` ainsi que les attributs et les méthodes de ces classes.

Question 6 : Quelle est la particularité de la classe `distributeur` ? Ecrire le code de la déclaration de cette classe.

Question 7 : Est-il nécessaire de définir un constructeur par recopie et un opérateur d'affectation pour les classes `pile` et `file` ? Expliquer.

Question 8 : Déclarer les classes `pile` et `file` et définir les méthodes requises.

Question 9 : On veut éviter qu'un utilisateur de la classe `pile` ou `file` puisse utiliser l'interface de la classe `liste` (cela risquerait de remettre en cause l'ordre dans lequel ces distributeurs distribuent ou éliminent leurs éléments...). Modifier votre code afin de prendre en compte cet aspect. Expliquer.

Pour implémenter la classe `file_de_priorite`, on veut utiliser un tableau de valeurs `int` alloué dynamiquement. Ce tableau aura une taille initiale de `10` et devra être agrandi, si nécessaire, au cours du stockage de nouveaux éléments.

Question 10 : Déclarer la classe `file_de_priorite` et définir toutes les méthodes requises, y compris celles qui permettent de gérer correctement la mémoire.

Pour une implémentation correcte, l'exécution du programme suivant provoque l'affichage ci-dessous :

```
#include "distributeur.h" #include <iostream>
void main(){
  //distributeur d; // provoque une erreur à la compilation
  std::cout<<"ESSAI PILE\n";
  pile p; p.put(4); p.put(9); p.put(2); p.put(7);
  std::cout<<p.item()<<"\n";  p.remove(); std::cout<<p.item()<<"\n";
  while(!p.empty()) p.remove();
  std::cout<<"ESSAI FILE\n";
  file f; f.put(4); f.put(9); f.put(2); f.put(7);
  std::cout<<f.item()<<"\n";  f.remove(); std::cout<<f.item()<<"\n";
  std::cout<<"ESSAI PRIORITE\n";
  file_de_priorite fp;
  fp.put(4); fp.put(9); fp.put(2); fp.put(7); fp.put(6); fp.put(1);
  std::cout<<fp.item()<<"\n"; fp.remove(); std::cout<<fp.item()<<"\n";
  file_de_priorite fp2=fp; fp2.remove(); std::cout<<fp2.item()<<"\n";
  std::cout<<fp.item()<<"\n"; fp=fp2; std::cout<<fp.item()<<"\n";
}
```

```
ESSAI PILE
7
2
ESSAI FILE
4
9
ESSAI PRIORITE
9
7
6
7
6
```

Exercice 2 - Le jeu des x erreurs. (changer de copie !)

Ces 2 programmes provoquent des erreurs de compilation. Expliquer pourquoi (2 à 3 lignes maximum par problème) et corriger.

Programme 1

```
class A{
  int attribut1;
public:
  explicit A(int i){ attribut1=i;}
  static int incrementeAttribut1(){
     return ++attribut1;
  }
};

void main(){
  int i;
  A a1;
  A a2=2;
  i=a2.incrementeAttribut1();
  A::incrementeAttribut1();
}
```

Programme 2

```
class A{
public:
  int attribut1;
  A(int i){ attribut1=i; }
};

class B:private A{
protected:
  int attribut2;
public:
  B(int i1,int i2):A(i1){ attribut2=i2; };
  void afficher(){
       std::cout<<attribut1<<" "<<attribut2;
  }
};

void main(){
  A a1=1; B b1(2,3);
  const B b2=b1;
  B *b3=&a1;
  a1.attribut1=4;
  b1.attribut1=5;
  b1.attribut2=6;
  b1.afficher();
  b2.afficher();
}
```