Final

Mercredi 25 juin 2014, de 8h00 à 10h00

Note : Documents cours et TD papier autorisés. La copie du voisin même papier, ne l'est pas.

1 Le summum (8pts)

Voici un foncteur :

```
struct monFoncteur{
      void operator()(int v){
            if (v>val )val =v;
      };
      monFoncteur(int& m) : val(m){};
private :
      int &val;
};
```

- Après l'avoir analysé, ecrivez une séquence utilisant l'algorithme *for_each*, pour appliquer le foncteur sur les éléments d'un vecteur. En vous assurant par initialisation que toute valeur comparée soit cohérente avec le contenu du vecteur.
- Expliquez le résultat souhaité à la suite de l'appel.
- Complétez le foncteur pour offrir la possibilité de travailler avec tout type de données possèdant l'opérateur ">". Vous ferez en sorte d'éviter des appels aux constructeurs par recopie dans le cas d'objet , tout en rendant la modification impossible.
- Nous souhaitons supprimer le paramètre du constructeur et définir ainsi un opérateur fonctionnel à 2 paramètres. Le premier sera la valeur correspondante à val. Ecrivez le foncteur "*monFoncteur2*" répondant à cet objectif.
- Ecrivez l'instruction d'utilisation de ce nouveau foncteur avec l'algorithme *for_each*. Pour ce faire vous utiliserez les nouvelles possibilités de **C++11** qui intègre, via le fichier "*functional*", les fonctionnalités créées par BOOST (fichier *boost/bind.hpp*)

Pour rappel :

- L'appel de l'algorithme est "***for_each(iterateurDebut, iterateurFin, Foncteur)***"
- Utilisation de **bind** :

```
int f(int a, int b) { return a + b; }
…
int i=bind(f, 1, 2)() ;
int x=3;
i= bind(f, _1, 5) (x);
i=bind(f, 5, _1)(x);
```

Il est possible de transmettre une référence constante ou non à l'aide de *cref* et re*f*

```
bind(g, ref(i), _1)(x);
bind(g, cref(42), _1)(x);
```

g est bien sûr une fonction adaptée.

2 La politique du pire (12pts)

Nous souhaitons créer un adaptateur d'objet pour ajouter des éléments dans un conteneur et retirer la valeur la plus lourde.

L'adaptateur fournira comme interface :

- Une fonction ***ajout*** de profil *void ajout(const int &)* permettant d'ajouter un entier dans le conteneur.
- Une fonction ***pire*** de profile *int pire()* permettant d'accéder à la valeur entière la plus lourde.
- Une fonction ***supprimerPire*** de profil *void suprimerPire()* permettant de supprimer la valeur la plus "lourde".

Pour ce faire nous allons utiliser la metaprogrammation et notamment la notion de **classe politique** et **classe hôte**. La classe hôte sera notre adaptateur :

```
template <class CONT, class Operations>
struct Adaptateur{
	Adaptateur(CONT &C) : Conteneur(C) {}
	void ajout(const int&val) {
		Operations::Ajout(Conteneur, val);
	}
	int pire() {
		return Operations::Pire(Conteneur);
	}
	void supprimerPire() {
		Operations::SupprimerPire(Conteneur);
	}
	CONT& Conteneur;
};
```

Le modèle des politiques applicables est :

```
template <class CONT>
struct ModelePoli {
    static inline void Ajout(CONT& C, const int& Valeur) {};
    static inline int Pire(CONT& C){return int(0);};  // int() est possible = 0
    static inline void SupprimerPire(CONT& C){};
};
```

- Ecrivez la classe politique pour un multiset d'entiers.

- Ecrivez la classe politique pour une priority_queue d'entiers.

- Voici différentes déclarations d'adaptateurs. On suppose qu'une politique est disponible pour le conteneur *set*. Y-a-t-il des instructions invalides ? Justifiez votre réponse.

```
  set<int> msi;
1 Adaptateur<int, set, ModelePoli <set> > AdaptS1(msi);
2 Adaptateur<set, ModelePoli <set<int> >> AdaptS2(msi);
3 Adaptateur<int, set, ModelePoli < set<int> >> AdaptS3(msi);
4 Adaptateur<set<int>, ModelePoli <int, set<int> >> AdaptS4(msi);
5 Adaptateur<set<int>, ModelePoli <set<int> >> AdaptS5(msi);
6 Adaptateur<set, ModelePoli < set>> AdaptS6(msi);
7 Adaptateur<set<int>, ModelePoli> AdaptS7(msi);
```

- Ecrivez une séquence définissant un multiset d'entiers et créant un adaptateur sur ce multiset.

- Modifiez les politiques pour travailler avec tout type possédant un opérateur "<".
- Modifiez la classe Hôte pour travailler avec ces politiques.
- Adapter la séquence de création de l'adaptateur.

Nous voulons travailler avec 2 adaptateurs de conteneurs dissemblables, stockant des données de même type, via la même variable de façon dynamique. Pour cela nous devons définir une interface mise en œuvre par chaque adaptateur.

- Ecrivez l'interface "*AdaptInterG*".
- Modifiez le modèle adaptateur pour qu'il soit utilisable via un pointeur sur un objet interface.
- La file de priorité fournit par défaut comme optimum l'objet de valeur maximale. Modifiez votre déclaration de file de priorité pour obtenir la valeur minimale.

Pour rappels :

Le type mutiset :

```
template < class T,                        // multiset::key_type/value_type
           class Compare = less<T>,        // multiset::key_compare/value_compare
           class Alloc = allocator<T> > // multiset::allocator_type
           > class multiset;
```

Le type priority_queue :

```
template < class T,                        // type valeur stockée
           class Container = vector<T>,  // conteneur support
           class Compare = less<typename Container::value_type>  // comparateur
           > class priority_queue;
```

Les opérations qui peuvent être utiles, disponibles pour un mutiset sont :

```
iterator begin();                          // premier élément dans un parcours
                                              séquentiel ordonné
reverse_iterator rbegin();                 // premier élément dans un parcours
                                              séquentiel ordonné inverse
iterator insert (const value_type& val);  // ajout d'un élément
iterator find (const value_type& val) const; // recherche d'un élément
void erase (iterator position              // destruction d'un élément
size_type erase (const value_type& val);  // destruction des éléments
                                              correspondant à la valeur val
```

Les opérations utiles qui peuvent être utiles, disponibles pour une file de priorité sont :

```
void push (const value_type& val);  // ajout d'un élément
const value_type& top() const;      // accès à l'élément le plus prioritaire
void pop();                         // suppression de l'élément prioritaire
explicit priority_queue (const Compare& comp = Compare(),
                         const Container& ctnr = Container());
                                    // un constructeur
```

Rappel :

« *std ::multiset* » est un type template « template<class T, class Op=less<T>, class A=allocator<T> class »
« *std ::multiset<T>* » avec T défini (exemple int), est une classe normale.