UNIVERSITÉ DE MONTPELLIER

UNIVERSITÉ DE MONTPELLIER - FACULTÉ DES SCIENCES ET TECHNIQUES
PLACE EUGÈNE BATAILLON - 34095 MONTPELLIER CEDEX 5

Licence 2 informatique – 2016/2017

HLIN302 – Examen n° 1

Programmation impérative avancée
Pascal GIORGI
[Durée 2h – Tous documents interdits]

1 Des cartes à jouer

Vous trouverez ci-dessous la signature d'une classe permettant de manipuler une carte de jeu (comme celles utilisées au poker). Une carte est représentée par deux attributs : une couleur (pique, coeur, carreau, trèfle) et une valeur comprise entre 1 et 13 (le valet vaut 11, la dame 12 et le roi 13).

carte.h

```
1  #ifndef __CARTE_H
2  #define __CARTE_H
3  #include <iostream>
4    class Carte {
5      public:
6        enum Couleur {pique, coeur, carreau, trefle};
7
8        Carte(Couleur, unsigned int);
9        Couleur getCouleur() const;
10       unsigned int getValeur() const;
11       void setCouleur(Couleur);
12       void setValeur(int);
13
14     private:
15       Couleur      col;
16       unsigned int val;
17   };
18   void afficheCarte(std::ostream&, const Carte&);//affiche une carte sur un flux
19 #endif
```

1. Expliquez (en 5 lignes maximum) à quoi sert le mot clé **const** utilisé aux lignes 9 et 10.
2. Expliquez (en 5 lignes maximum) à quoi sert le mot clé **private** utilisé à la ligne 14. Vous motiverez également l'intérêt de cette caractéristique pour la définition de la classe Carte (vous pouvez utiliser un exemple),
3. La ligne 18 définit *afficheCarte* . Est-ce que cette déclaration correspond à une méthode ou une fonction ?
4. Donner le code complet de la surcharge de l'opérateur d'affichage sur un flux pour la classe Carte (vous devez utiliser obligatoirement *afficheCarte*). Vous illustrerez l'utilisation de cet opérateur par appel explicite et implicite dans un programme.
5. Expliquez (en 5 lignes maximum) pourquoi la construction d'un tableau de cartes (non initialisé, ex. *Carte T*[4]) n'est pas possible. Expliquer ce qu'il faut modifier dans la classe pour rendre cela possible.

Nous souhaitons maintenant créer une classe permettant de stocker une pile de carte au travers d'un tableau. Cette classe doit permettre **UNIQUEMENT** de construire une pile de cartes vide, d'ajouter une carte sur le dessus de la pile, et de récupérer la carte au dessus de la pile (en l'enlevant). Votre classe ne devra pas fournir d'accesseurs ni d'autres méthodes.

6. Donner la signature de la classe *PileCartes* sachant que la taille de la pile doit être indéterminée à la compilation. Attention : votre classe devra gérer complètement la mémoire ; vous donnerez **UNIQUEMENT** les signatures de toutes les méthodes nécessaires avec un commentaire pour expliquer leur fonctionnement.
7. Donnez le code complet de tous les constructeurs de votre classe.

2 Des listes circulaires

Nous avons vu en cours l'implantation d'une classe liste doublement chaînée. Dans cet exercice, nous allons reprendre les principes de cette classe pour proposer une classe de liste circulaire. Afin de simplifier l'exercice, nous ne considérerons que le cas d'une liste simplement chaînée (les éléments de la liste n'ont que des successeurs, pas de prédécesseurs). L'idée de notre liste circulaire est que chaque élément interne de la liste à toujours un successeur. En particulier, le successeur du dernier élément de la liste et le premier élément. Nos listes devront gérer la mémoire avec des copies en profondeur. Voici les signatures des modèles de classe permettant de représenter nos listes circulaires.

Fichier `node.h`

```
#ifndef __NODE_H
#define __NODE_H
template<typename T>
class Node {
  private:
    T        val;
    Node*    next;
  public:
    Node(const T&);
    T getVal() const;
    Node* getNext() const;

    void setVal(int);
    void setNext(Node*);
};
#include "node.tpl"
#endif
```

Fichier `liste-circ.h`

```
#ifndef __LISTE_CIRC_H
#define __LISTE_CIRC_H
#include "node.h"
template<typename T>
class ListeCirc {
  private:
    Node<T>* first;
    Node<T>* getLast();
  public:
    ListeCir();
    ~ListeCirc();
    ListeCirc(const ListeCirc&);
    const ListeCirc& operator=(const ListeCirc&);

    void ajoutTete(const T&);
    void avanceTete();
    T getTeteValeur() const;
};
#include "liste-circ.tpl"
#endif
```

La méthode *ajoutTete* ajoute un élément en tête de la liste, en garantissant la conservation de la structure de liste circulaire. La méthode *avanceTete* déplace la tête sur son successeur dans la liste. La méthode *getTeteValeur* permet de récupérer la valeur de l'élément en tête de liste. La méthode privée *getLast*() permet de récupérer un pointeur sur le dernier élément de la liste.

1. Donnez une représentation mémoire (sur une machine 64-bits) pour les 3 listes circulaires suivantes après l'exécution de toutes les instructions (en spécifiant les tailles en octets et les valeurs stockées) :
   - — *ListeCirc*<**int**> *L1*;
   - — *ListeCirc*<**double**> *L2*; *L2.ajoutTete* (3.14);
   - — *ListeCirc*<*Carte*> *L3*; *L3.ajoutTete* (*Carte*(*pique* ,11)); *L3.ajoutTete* (*Carte*(*coeur* ,3));
2. Donnez le code complet de la méthode *getLast*. Vous spécifierez également la complexité de cette méthode en fonction du nombre d'éléments dans la liste.

3. Donnez le code complet de la méthode *ajoutTete* du modèle de classe *ListeCirc* (en utilisation la séparation des signatures et des implantations, i.e. fichier .tpl). Vous spécifierez également la complexité de cette insertion en fonction du nombre d'éléments dans la liste. Pensez-vous que cela peut être amélioré ? si oui comment (vous pouvez ajouter des attributs).

3 Une pile de cartes avec remise

Nous allons maintenant mixer nos cartes et notre liste circulaire pour proposer une classe gérant une pile de cartes avec remise en dessous de pile. L'objectif de cette classe est de représenter une pile de cartes comme précédemment mais lorsqu'une carte est enlevée du sommet de la pile, on la remet tout en bas de la pile (comme on le fait souvent dans certain jeu de cartes).

1. En utilisant la classe *Carte* et le modèle de classe *ListeCirc* proposer la signature de la classe *PileCarteRem*. Comme précédemment, cette classe doit permettre la construction d'une pile de cartes vide, d'ajouter une carte sur le dessus de la pile, et de récupérer la carte au dessus de la pile (en la déplaçant en fin de pile).
2. Donner le code complet des méthodes *empile* et *depile* permettant respectivement :
   - — d'ajouter une carte en haut de la pile
   - — de déplacer la carte du haut de la pile et fin de pile, et de retourner une copie de cette carte.
3. Expliquez pourquoi cette classe n'a pas besoin de définir le destructeur, le constructeur par copie et l'opérateur d'affectation.