**Contrôle Continu AP4A - Vendredi 30 Avril 2021 - durée 1 heure**

**I** (6 pts) **-** On souhaite écrire une fonction swap en C++, pour permuter le contenu de 2 variables qui peuvent être des objets ou des types élémentaires. Par exemple, pour tester le swap on souhaite pouvoir écrire :

{

 int i = 2, j = 3;

 cout << "i " << i << " j " << j << endl;

 swapX(i, j);

 cout << "i " << i << " j " << j << endl;

 Point p1(1,2), p2(3,4);

 cout << "p1 : " << p1 << " p2 : " << p2 << endl;

 swapY(p1, p2);

 cout << "p1 : " << p1 << " p2 : " << p2 << endl;

 vecteur<int> v1(3), v2(3);

 v1[0] = 1; v1[1] = 2; v1[2] = 3;

 v2[0] = 4; v2[1] = 5; v2[2] = 6;

 cout << "v1 : " << v1 << " v2 : " << v2 << endl;

 swapZ(v1, v2);

 cout << "v1 : " << v1 << " v2 : " << v2 << endl;

}

1) (4 pts) Considérons les implémentations suivantes d’une fonction swap (fonction seule), indiquer lesquelles sont correctes ou incorrectes, syntaxiquement (erreur compilation) et sémantiquement (erreur exécution), expliciter les raisons de leur validité ou non, notamment en relation au programme de test.

void **swap1**(int i, int j) {

 int k = i;

 i = j;

 j = k;

}

void **swap2**(int& i, int& j) {

 int k = i;

 i = j;

 j = k;

}

void **swap3**(int\* i, int\* j) {

 int k = \*i;

 \*i = \*j;

 \*j = k;

}

void **swap4**(Point p1, Point p2) {

 Point k = p1;

 p1 = p2;

 p2 = k;

}

void **swap5**(Point& p1, Point& p2) {

 Point k = p1;

 p1 = p2;

 p2 = k;

}

void **swap6**(Point\* p1, Point\* p2) {

 Point k = p1;

 p1 = p2;

 p2 = k;

}

2) (2 pts) Donner une fonction swap pour la classe vecteur. Quel mécanisme peut-on utiliser pour n’écrire qu’une seule et unique fonction swap() pour tout type de donnée et qui fonctionnera pour le test ci-dessus. Si ce mécanisme existe, écrivez la fonction swap().

**II** (14 pts) – On considère ici la classe Liste telle que codée en TD, en récursif :

template <class T>

class Liste {

 T val; // tete de liste = element

 Liste\* reste; // suivi d'une liste

public:

…

}

1) (3 pts) Considérons maintenant la classe Ensemble telle que codée en TD par héritage privé de la classe Liste et notamment sa fonction appartient() :

template <class T>

class Ensemble : private Liste<T> {

public:

...

 // renvoie 0(=FAUX) si n'appartient pas

 // renvoie la position + 1 dans la liste sinon (!=0)

 int **appartient**(const T& x) {

 for(int i = 0; i < this->longueur(); i++){

 if( (\*this)[i] == x ){

 return i + 1;

 }

 }

 return 0;

 }

...

}

1. (2 pts) Rappeler le profil de l’opération d’accès operator[] de la classe Liste et son corps.
2. (1 pts) En quoi cette fonction n’est-elle pas appropriée pour le parcours de la liste via une boucle « for » comme dans la fonction appartient() ci-dessus. Expliquer pourquoi.

2) (7 pts) Pour remédier au problème que vous venez d’expliciter, on souhaite définir une sous-classe ListeIterateur (incluse à l’intérieur) de la classe Liste pour parcourir le contenu de la liste (telle que codée en récursif) :

template <class T>

class Liste {

…

public:

…

 class ListeIterateur {

 …

 public:

 ListeIterateur(...) …

 void **reset**() …

 ... **hasNext**()…

 ... **getNext**()…

 };

};

et l’utiliser par exemple de la façon suivante :

{

 Liste<int> l1 ;

 ...

 Liste<int>::ListeIterateur iter(l1);

 cout << "Premier passage l1 : " << endl;

 iter.reset();

 while (iter.hasNext()) {

 int& val = iter.getNext() ;

 cout << "Element suivant : " << val << " " << endl;

 }

 cout << "Deuxième passage l1 : " << endl;

 iter.reset();

 while (iter.hasNext()) {

 int& val = iter.getNext() ;

 cout << "Element suivant : " << val << " " << endl;

 }

}

La fonction reset() permet de réinitialiser le parcours. Expliquer pourquoi cet itérateur permettra de résoudre le problème mentionné au point précédent. Donner le codage complet (interface et implémentation) de la sous-classe ListeIterateur. Attention de bien énumérer tous les éléments de la liste.

3) (2 pts) Donner une implémentation de la méthode appartient() avec utilisation de l’itérateur dans le cas de l’implémentation de la classe Ensemble par héritage privé de la classe Liste, pour remédier au problème précédent.

4) (2 pts) Enfin il peut être utile de fournir un itérateur pour la classe Ensemble elle-même, accessible publiquement. Ecrivez une sous-classe EnsembleIterateur incluse dans la classe Ensemble qui réutilise la sous-classe ListeIterateur et permettra d’écrire des parcours du type :

{

 Ensemble<int> e3;

 ...

 Ensemble<int>::EnsembleIterateur iter(e3);

 iter.reset();

 while (iter.hasNext()) {

 int& val = iter.getNext() ;

 cout << "Element suivant : " << val << " " << endl;

 }

}