

Trimestre Été, 2008

Mohamed Lokbani

IFT1166 – Examen Final Solutionnaire –

Inscrivez tout de suite votre nom et le code permanent.

Nom: _____ | Prénom(s): _____ |

Signature: _____ | Code perm: _____ |

Date : mercredi 9 juillet 2008

Durée : 3 heures (de 17h30 à 20h30)

Local : Z-317 ; Pavillon Claire McNicoll

Directives:

- Toute documentation est permise.
- Calculatrice **non** permise.
- Répondre directement sur le questionnaire.
- Les réponses **doivent être brèves, précises, claires et nettement présentées.**

1. _____ /20 (1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8)

2. _____ /15 (2.1, 2.2)

3. _____ /20 (3.1, 3.2, 3.3, 3.4, 3.5)

4. _____ /20 (4.1, 4.2)

5. _____ /25 (5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7)

Total: _____ /100

Directives officielles

* Interdiction de toute communication verbale pendant l'examen.

* Interdiction de quitter la salle pendant la première heure.

* L'étudiant qui doit s'absenter après la première heure remettra sa carte d'étudiant au surveillant, l'absence ne devant pas dépasser 5 minutes. Un seul étudiant à la fois peut quitter la salle.

* Toute infraction relative à une fraude, un plagiat ou un copiage est signalée par le surveillant au directeur de département ou au professeur qui suspend l'évaluation.

F.A.S

Exercice 1 (20 points)

1.1 Un destructeur est appelé dans le cas (cochez une ou plusieurs réponses) :

- a.** d'une variable locale qui quitte sa portée.
- b.** d'un appel à « delete ».
- c.** d'un appel à « delete [] ».
- d.** d'aucune des réponses précédentes.

1.2 Peut-on détruire une instance d'une classe si le destructeur de la classe est déclaré privé?

- (a) oui, il est possible de la détruire | **(b) non, il n'est pas possible de la détruire**

Pourquoi ?

Il y aura toujours appel au destructeur soit dès la fin de la portée d'une variable ou bien suite à un appel explicite à « delete ». De ce fait, il n'est pas possible de déclarer le destructeur « private » (voir aussi question 1.1).

1.3 Définissez ce qu'est « la déclaration d'une classe ».

Déclarer une classe: déclarer les membres « données » et les méthodes

Déclarer membres données: juste les citer.

Déclarer une méthode membre (donc une fonction): écrire son en-tête (ou prototype) .

1.4 Les membres d'une classe dérivée ont-ils accès à tous les membres de sa classe de base quelque soit leurs droits d'accès?

- (a) oui, ils ont accès | **(b) non, ils n'ont pas accès**

Pourquoi ?

Ils n'auront accès qu'aux membres dit « public ». « private » personne n'a droit d'accès par définition. « protected », les classes dérivées ont accès aux membres « protected » des classes de base, mais pas l'inverse.

1.5 Le mot clé réservé « inline » devant une fonction est juste à titre indicatif pour le compilateur. Celui-ci peut en effet décider en dernier lieu de l'utiliser ou non.

Pourquoi ?

Le qualificatif « inline » recommande au compilateur de faire une copie du code de la fonction en place. Cependant le compilateur peut prendre la décision de "désactiver" le qualificatif « inline » d'une fonction s'il estime qu'il perdra moins de temps à y accéder via la table des fonctions que de recopier son code.

1.6 Écrire le prototype de la fonction « test » qui retourne un pointeur sur un double et qui accepte les 3 arguments suivants : une chaîne de caractères du type « string » passée par référence, un caractère passé par valeur et initialisé par défaut avec la lettre « w » et finalement un entier passé par pointeur et initialisé par défaut avec la valeur « NULL ».

```
double* test(string& p, char x = 'w', int* z = NULL);
```

1.7 Citez les appels possibles à la fonction « test » précédemment définie (Q. 1.6).

Si `ptrs` est du type `string`, `a` du type `char` et `y` du type `int*`, nous avons alors :

```
test(ptrs);  
test(ptrs, x);  
test(ptrs, x, z);
```

1.8 Soit les fichiers suivants :

en-tête	source	objet	exécutable
a.h	a.cpp	a.o	
	test.cpp	test.o	test.exe

Nous avons déclaré dans le fichier « a.h » la méthode « affiche ». Cette dernière est appelée dans la fonction « main » définie dans le fichier « test.cpp ».

a. Tout en expliquant brièvement votre réponse, que faudra-t-il ajouter dans le fichier « test.cpp » pour que ce dernier compile sans problème?

Le fichier d'en-tête « a.h » qui contient le prototype de la méthode « affiche » nécessaire au compilateur, pour que ce dernier ne génère par une erreur de syntaxe.

b. Peut-on compiler le fichier « test.cpp » sans faire appel au fichier « a.cpp »?

(a) oui, c'est possible | (b) non, ce n'est pas possible

Pourquoi ?

Oui. La compilation n'a besoin que du fichier d'en-tête comme il a été mentionné dans la précédente réponse.

c. Écrire la ligne de commandes (appelant à gcc) qui permet de générer le fichier exécutable à partir des fichiers objets.

```
g++ -o test.exe test.o a.o
```

Exercice 2 (15 points) Soit la classe « A » :

```
01 | class A{
02 |     public:
03 |         int x;
04 |         A() {
05 |             x = 5;
06 |         }
07 |     };
```

2.1 Combien de fois est appelé le constructeur de la classe « A » dont la signature est « A() » dans le fragment de code suivant :

```
01 |     A u;
02 |     A v;
03 |     A *x;
04 |     x = new A();
05 |     A y = u;
06 |     A *z = new A();
```

Nombre de fois : 4

Pourquoi?

A u; // 1^{er} appel

A v; // 2^e appel

A *x; // c'est juste une déclaration. Il n'y a pas d'appel au constructeur.

x = new A(); // 3^e appel

A y = u; // appel au constructeur de copie.

A *z = new A(); // 4^e appel

2.2 Que va afficher en sortie le programme suivant qui compile et s'exécute correctement :

```
01 | #include <iostream>
02 | #include <string>
03 |
04 | using namespace std;
05 |
06 | class A{
07 | public:
08 |     int x;
09 |     A() { x = 5;}
10 | };
11 | class B: public A {
12 | public:
13 |     static int x;
14 |     B() { x++;}
15 |     B(int i) { x = x+i;}
16 |     B(string s) {x--;}
17 | };
18 | int B::x=10;
19 |
20 | int main() {
21 |     B b1;
22 |     B b2(2008);
23 |     B b3("IFT1166");
24 |     cout << b1.x << " : " << b2.x << " : " << b3.x << endl;
25 |     return 0;
26 | }
27 |
```

Affichage produit avec vos explications :

Affichage en sortie :

2018 : 2018 : 2018

Explications :

À noter que « x » de « B » est « static ». Il est donc unique. Il est partagé par toutes les instances de « B ».

« B b1; » fait appel au constructeur sans argument « B() ». Ce dernier incrémente « x » de « B ». Au départ « x » valait « 10 ». Suite à cette opération, « x » vaut maintenant « 11 ».

« B b2(2008); » fait appel au constructeur avec argument « B(int) ». Ce dernier additionne à « x » la valeur de l'argument. Au départ « x » valait « 11 ». Suite à cette opération, « x » vaut maintenant « 2019 ».

« B b3("IFT1166"); » fait appel au constructeur avec argument « B(string) ». Ce dernier décrémente la valeur de « x ». Au départ « x » valait « 2019 ». Suite à cette opération, « x » vaut maintenant « 2018 ».

La ligne 24 permet d'afficher la valeur de « x » via 3 instances. Or comme il a été déjà mentionné, « x » est unique et il est partagé par toutes les instances « b1, b2, b3 ». Ce qui explique pourquoi la valeur de x, affichée en sortie, est la même peu importe l'instance qui fait appel à « x ».

Exercice 3 (20 points) Soit les définitions suivantes :

```
01
02     class UneInterface {
03     public:
04         virtual void methodeX(int c)=0;
05     };
06
07     class A {
08     public:
09         void methodeX(int c) {}
10     };
11
12     class B: public UneInterface {
13     public:
14         void methodeY(int c) {}
15         void methodeX(int c) {}
16     };
17
18     class C: public B {
19     public:
20         void methodeZ(int c) {}
21     };
22
```

Parmi les groupes suivants d'instructions lesquels compilent correctement ?

3.1 `UneInterface* x = new A();`
`x->methodeX(10);`

correct

incorrect

Pourquoi ?

Une instance de « A » ne peut être convertie à « UneInterface » car il n'y a aucune relation entre les deux classes.

3.2 `UneInterface* y = new B();`
`y->methodeY(10);`

correct

incorrect

Pourquoi ?

L'instruction « `UneInterface* y = new B();` » est correcte car « B » hérite de « UneInterface ».

L'appel suivant « `y->methodeY(10)` » pour qu'il soit correct, il faut que « `methodeY` » existe et dans « B » et dans « UneInterface ». Ce qui n'est pas le cas.

La classe « UneInterface » a besoin de connaître l'existence de la fonction « `methodeY` ». Or cette méthode n'a été définie que dans la classe B.

3.3 `UneInterface* z = new B();`
 `z->methodeX(10);`

correct

incorrect

Pourquoi ?

Pour les mêmes raisons expliquées en « 3.2 » qu'en est-il de la méthode « `methodeX` » ?

La classe « `UneInterface` » a besoin de connaître l'existence de la méthode « `methodeX` ». Cette méthode a été définie dans les deux classes « `UneInterface` » et « `B` ». L'appel est donc correct.

3.4 `UneInterface* w = new C();`
 `w->methodeZ(10);`

correct

incorrect

Pourquoi ?

« `C` » hérite de « `B` » qui hérite de « `UneInterface` », donc ok. Par contre la méthode « `methodeZ` » n'est disponible que dans la classe « `C` ». D'où l'erreur pour les mêmes raisons que 3.2.

3.5 `UneInterface* q = new C();`
 `q->methodeX(10);`

correct

incorrect

Pourquoi ?

La méthode « `methodeX` » a été définie dans la classe « `B` ». La classe « `C` » hérite des propriétés de la classe « `B` ». De ce fait « `methodeX` » est connue aussi de « `C` ». Puisque l'existence de la méthode est connue des classes « `UneInterface` » et « `C` », il n'y a pas donc d'erreur.

Exercice 4 (20 points) Un compteur est un objet défini par les propriétés suivantes :

- Il possède une valeur entière, positive ou nulle, nulle à sa création. Il ne peut varier que par pas de 1 (incréméntation).
- Il possède aussi deux méthodes « incremente » et « getValeur » pour respectivement incrémenter la valeur entière et obtenir la valeur courante.

4.1 Écrire la définition de la classe « Compteur » (y compris la définition des méthodes).

```
class Compteur {
private:
    int valeur;
public:
    Compteur(int x=0) : valeur(x) {}
    void incremente() {valeur++;}
    int getValeur() { return valeur;}
};
```

4.2 Une pièce de monnaie est définie par ses deux cotés « pile » et « face ». Soit « UnExemple » une fonction sans arguments et sans valeur de retour qui contient les instructions nécessaires pour réaliser ce qui suit :

- Lancer aléatoirement 100 fois une pièce de monnaie.
- En supposant que nous avons une chance sur deux d’avoir l’une des deux faces possibles (pile ou face avec donc 50/50 de chance) lors de chaque tirage, les compteurs sont mis à jour pour les cotés pile et face de la pièce en utilisant pour cela la classe « Compteur » de 4.1. Nous allons supposer qu’une valeur aléatoire égale à 0 correspond au coté « pile » et si elle est égale à 1 c’est le coté « face ».
- Afficher sur la sortie standard le résultat obtenu (le nombre de tirages « pile » et celui de « face ») après les 100 tirages.

Vous pouvez supposer que nous avons inclus dans le programme tous les fichiers d’en-tête nécessaires.


```

void UnExemple() {

    // Pour générer des nombres différents à chaque fois
    // sinon mettre une valeur égale à 1 à la place de NULL

    srand(static_cast<unsigned>(time(NULL)));

    Compteur Pile;
    Compteur Face;

    for (int i=0; i<100; i++){

        // Retourne une valeur aléatoire du type int entre 0 et 2
        // 2 n'est pas inclus. Donc uniquement 0 ou 1.

        int x = rand()%2;

        if (x==0)
            Pile.incremente();
        else
            Face.incremente();

    }

    cout << "Nombre de Piles: " << Pile.getValeur() << endl;
    cout << "Nombre de Faces: " << Face.getValeur() << endl;

}

```

Exercice 5 (25 points)

5.1 Écrivez le code de la fonction « ConvertirMinus » dont le prototype est :
`void ConvertirMinus(string& mot);`

Les caractères de l'argument « mot » sont convertis en minuscule en utilisant la fonction « tolower » définie dans la librairie standard du langage et dont le prototype est : « int tolower(int c); ». Pour obtenir la taille de « mot », vous pouvez faire appel à la méthode « length() » définie pour le type « string ».

```
void ConvertirMinus(string& mot){
    for (unsigned int i =0; i< mot.length(); i++)
        mot[i] = tolower(mot[i]);
}
```

5.2 Écrivez le code de la fonction « SwapElements » dont le prototype est :
`void SwapElements(char tab[], int i, int j);`

Cette fonction permet d'échanger le contenu de « tab[i] » avec celui de « tab[j] ».

```
void SwapElements(char t[], int i, int j){
    char temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}
```

5.3 Écrivez le code de la fonction « ConvertirString » dont le prototype est :
`char* ConvertirString(string mot);`

Cette méthode permet de convertir l'argument mot du type « string » vers « char* ». C'est l'équivalent de la méthode « c_str() » dont l'utilisation dans ce cas est interdite.

```
char* ConvertirString(string mot){
    int taille = mot.length();
    char* nouveau = new char[taille+1];
    for (int i =0; i< taille; i++)
        nouveau[i] = mot[i];
    nouveau[taille] = '\0';
    return nouveau;
}
```

Pour cette question, on suppose que les méthodes décrites dans les questions « 5.1 », « 5.2 » et « 5.3 » sont déjà codées.

5.4 Écrivez le code de la méthode « SortString » dont le prototype est :

```
string SortString(string mot);
```

Cette méthode retourne l'argument « mot » trié. Par exemple, pour le mot « billet », la méthode retourne « beillt ». Utilisez l'algorithme de tri, « Tri par sélection » déjà vu en cours, qu'il faudra adapter à ce problème.

```
void triParSelection(char* table) {
    int min;
    int taille = strlen(table);
    for (int i=0; i < taille - 1 ; i++) {
        min = i;
        for (int j = i+1 ; j < taille ; j++)
            if (table[j] < table[min])
                min = j;
        if (min != i)
            SwapElements(table, i, min);
    }
}

string SortString(string mot) {
    ConvertMinus(mot);
    char* test = ConvertirString(mot);
    triParSelection(test);
    return string(test);
}
```

Pour ce qui suit, on suppose que les méthodes décrites dans les questions « 5.1 », « 5.2 », « 5.3 » et « 5.4 » sont déjà codées. Par ailleurs, vous avez en votre possession une variante des classes « Cellule » et « Liste » déjà utilisées en cours :

```
01 |
02 | class Cellule {
03 |     string valeur;
04 |     Cellule* prochain;
05 | public:
06 |     // Constrcuteur
07 |     Cellule(string v, Cellule* suivant);
08 |     // Obtenir la valeur de son prochain
09 |     Cellule* get_prochain();
10 |     // Fixer la valeur de son prochain
11 |     void set_prochain(Cellule* suivant);
12 |     // Obtenir la valeur associée à la cellule
13 |     string get_valeur();
14 | };
15 |
16 | class Liste {
17 |     Cellule *tete, *queue;
18 |     int longueur;
19 | public:
20 |     // Constructeur
21 |     Liste();
22 |     // Ajoute un élément au début
23 |     void ajout_tete(string x);
24 |     // Ajoute un élément à la fin
25 |     void ajout_queue(string x);
26 |     // Ajoute une autre liste à la fin
27 |     void ajout_liste(Liste *l);
28 |     // Affiche le contenu de la liste
29 |     void affiche();
30 |     // Efface tout le contenu de la liste
31 |     void efface();
32 |     // Retourne le nombre d'éléments dans la liste
33 |     int getLongueur();
34 |     // Retourne le ième élément de la liste
35 |     string getElt(int);
36 | };
37 |
```

La variable « valeur » de la classe « Cellule » est maintenant du type « string » au lieu de « int ». Nous avons ajusté les différentes méthodes des classes « Cellule » et « Liste » pour tenir compte de cette variante. Par ailleurs, nous avons ajouté les deux méthodes « int getLongueur() » et « string getElt(int) » dans la classe « Liste ». Nous allons supposer que toutes les méthodes des deux classes sont déjà définies.

5.5 Écrivez la méthode « int getLongueur() »

```
int Liste::getLongueur() {
    return longueur;
}
```

5.6 Écrivez la méthode « string getElt(int) ». Cette méthode retourne une chaîne vide si l'index spécifié ne correspond à aucun élément.

Note: le premier élément est à la position 0.

```
string Liste::getElt(int i){
    string s = "";
    if (index >= longueur || index < 0)
        return s;
    Cellule* courant = tete;
    while(index>0 && courant!=NULL){
        courant = courant->get_prochain();
        index--;
    }
    if (courant!=NULL)
        s = courant->get_valeur();
    return s;
}
```

5.7 Complétez le programme par une fonction « main » qui doit réaliser ce qui suit :

- Initialise la variable « MOTS » du type « Liste » avec les mots (dans cet ordre) « ami », « Mai », « Bonjour », « Brisavion » et « BorisVian ».

- Retourne une liste « RESULTAT » où se trouve un exemplaire de chacune des valeurs de « MOTS » sauf celles qui sont une anagramme d'un élément déjà dans « RESULTAT ». Pour mémoire une anagramme d'un mot est un mot formé des mêmes lettres que le premier mais dans un ordre différent. Par exemple le mot « AMI » est une anagramme de « MAI ».

Vous devez éviter les duplications! Ainsi donc, pour le précédent exemple, vous allez produire comme résultat la liste de mots : « ami », « Bonjour », « Brisavion ».

- Et affiche en sortie les listes « MOTS » et « RESULTAT », comme suit :

La liste des mots: ami Mai Bonjour Brisavion BorisVian

La liste des anagrammes: ami Bonjour Brisavion

Note: si vous le jugez nécessaire, libre à vous de définir d'autres listes pour manipuler vos données.

```

int main() {

    Liste MOTS;

    MOTS.ajout_queue("ami");
    MOTS.ajout_queue("Mai");
    MOTS.ajout_queue("Bonjour");
    MOTS.ajout_queue("Brisavion");
    MOTS.ajout_queue("BorisVian");

    Liste ANAGRAM;
    Liste RESULTAT;

    RESULTAT.ajout_queue(MOTS.getElt(0)); // On ajoute de facto l'élément 0.

    bool flag;

    int tailleMots = MOTS.getLongueur();

    for (int i=1 ; i<tailleMots; i++) {
        string element = MOTS.getElt(i);
        string s = SortString(element);
        flag = true;
        int tailleANAGRAM = ANAGRAM.getLongueur();
        for (int j=0 ; j<tailleANAGRAM; j++) {
            string data = ANAGRAM.getElt(j);
            if (s==data){
                flag=false; // éléments sont identiques
                break; // pas besoin de parcourir le reste de la liste
            }
        }
        if (flag){
            ANAGRAM.ajout_queue(s);
            RESULTAT.ajout_queue(element);
        }
    }

    cout << "La liste des mots: ";
    MOTS.affiche();

    cout << "La liste des anagrammes: ";
    RESULTAT.affiche();

    return 0;
}

```

À noter :

L'utilisation des deux listes « ANAGRAM » et « RESULTAT », en plus de la liste « MOTS », nous permet d'éviter de faire des appels intempestifs à « SortString ».

Bonnes vacances à toutes et à tous.