POLYTECHNIQUE MONTRÉAL

Département de Génie Informatique et Génie Logiciel

Cours INF1010 : Programmation orientée objet (Hiver 2018)

CONTRÔLE PÉRIODIQUE
CORRIGÉ

DATE : Le vendredi 23 février 2018

HEURES : de 18h30 à 20h20

DUREE : 1h50

PONDERATION : 30%

LIEU:

Section 1: B-418.
Section 2: A-416 / B-415.
Section 3: B-314 / B-315.
Section 4: A-410 / A-416.

Documentation, calculatrice et appareils électroniques sont interdits.

Vous devez répondre sur le cahier. Soignez votre écriture.

Cet examen comprend 5 questions sur 10 pages pour un total de 20 points.

Question 1 – Questions en vrac (4 points)

Soient les classes suivantes :

```
class Jouet {
public:
    Jouet(){};
private:
    double prix_;
};

class Personne {
public:
    Personne(/* CODE MANQUANT */) {
    /* CODE MANQUANT */ };
private:
    Jouet & monJouet_;
};
```

```
class Employe: public Personne{
public:
     ....
     void afficher();
private:
    ......
};
```

(a) **(½ point)** L'espace mémoire dynamique est réservé sur la pile ◯ Vrai √ **Faux**

(b) **(½ point)** Transmettre à une fonction globale (ou à une méthode) un paramètre objet par référence constante permet d'améliorer la vitesse d'exécution d'un programme. √ **Vrai** ◯ Faux

(c) **(½ point)** Soient les déclarations suivantes, écrire les instructions manquantes pour exécuter la méthode `afficher()` de la classe `Employe`.

```
Personne * ptrPersonne = new Employe ();
Employe * ptrEmploye;
```

Solution:

```
ptrEmploye = static_cast<Employe* > ptrPersonne;
ptrEmploye ->afficher();
```

(d) **(½ point)** Les classes `Personne` et `Jouet` ont-elles une relation de composition, d'agrégation ou d'héritage ? ◯ Composition √ **Agrégation** ◯ Héritage

(e) **(½ point)** Écrire la définition et l'implémentation du constructeur de la classe `Personne`.

Solution:

```
Personne(Jouet& j) : monJouet_(j) {};
```

(f) **(1½ point)** Soient les signatures des fonctions globales suivantes et les déclarations suivantes :

```
void tester(Jouet *p);  // A
void tester(Jouet **p); // B
void tester(Jouet &p);  // C
void tester(Jouet p);   // D

Jouet * p  = new Jouet();
Jouet q;
Jouet ** t = new Jouet * [10];
```

Pour chacune des instructions ci-dessous, indiquez laquelle ou lesquelles des fonctions globales présentées précédemment pourrai(ent) être appelée(s). Si aucune de ces fonctions globales n'était appelée par l'instruction, indiquez « aucune ».

1. `tester(new Jouet());`
2. `tester(new Jouet [10]);`
3. `tester(new Jouet *[10]);`
4. `tester(p);`
5. `tester(*p);`
6. `tester(q);`
7. `tester(&q);`

Solution:

1. `tester(new Jouet());` appelle la fonction A
2. `tester(new Jouet[10]);` appelle la fonction A
3. `tester(new Jouet*[10]);` appelle la fonction B
4. `tester(p);` appelle la fonction A
5. `tester(*p);` appelle la fonction C ou la fonction D
6. `tester(q);` appelle la fonction C ou la fonction D
7. `tester(&q);` appelle la fonction A

Question 2 – Concepts de base, surcharge des opérateurs (5 points)

Vous êtes embauché par la compagnie « Désinformatique SA » et on vous demande de maintenir une classe `MathVector` pour les vecteurs (la notion mathématique). Malheureusement, la version courante contient des erreurs de compilation et elle est incomplète. Avant de corriger le code, vous devez rafraîchir vos connaissances sur la surcharge des opérateurs et sur les notions mathématiques.

Soient deux vecteurs $\overrightarrow{u} = (a, b)$ et $\overrightarrow{v} = (c, d)$, et un scalaire k,

Addition	$\overrightarrow{u} + \overrightarrow{v} = (a + c, b + d)$
Multiplication par un scalaire	$k * \overrightarrow{u} = k * (a, b) = (ka, kb)$
Produit scalaire	$\overrightarrow{u} * \overrightarrow{v} = ac + bd$

Soit le code suivant :

```
class MathVector {
public:
    MathVector(int x = 0, int y = 0):x_(x),y_(y){};

    MathVector operator+(const MathVector& droite);
    MathVector& operator+=(const MathVector& droite);
```

```
    int operator*(const MathVector& droite);
    MathVector operator*(const int& gauche, const MathVector& droite);

    friend ostream& operator<<(ostream o, const MathVector& droite) {
        return o<<"<"<<droite.x_<<","<<droite.y_<<">";
    }

private:
    int x_;
    int y_;
};

int main() {
    MathVector v1(4,5);
    MathVector v2(1,-2);

    cout<<"Addition: "<<v1+v2<<endl;
    cout<<"Augmentation: "<<(v1+=v2)<<endl;

    cout<<"Multiplication par scalaire: "<<5*v2<<endl;
    cout<<"Produit scalaire: "<<v1*v2<<endl;

    return 0;
}
```

(a) **(1 point)** Répondez par vrai ou faux aux questions suivantes :

i. En ayant défini `operator+`, il n'est pas nécessaire de définir `operator+=`.
◯ Vrai √ **Faux**

ii. Pour la classe `MathVector`, il n'est pas nécessaire de déclarer `operator<<` comme `friend`. ◯ Vrai √ **Faux**

iii. En retournant une référence d'un objet `ostream`, on peut appeler `operator<<` en cascade. √ **Vrai** ◯ Faux

iv. L'opérateur `operator*(const int & gauche, const MathVector & droite)` ne garantit pas la propriété de la commutativité, par conséquent, on doit redéfinir l'opérateur `operator*(const int & droite)`. √ **Vrai** ◯ Faux

(b) **(1 point)** Dans le code de la classe, il y a deux erreurs d'écriture de code. Trouvez-les, corrigez-les et expliquez ces erreurs.

Solution:

1. La fonction `operator*(int,MathVector)` doit être globale. Elle ne peut pas être une méthode. Étant donné que la fonction accède aux attributs privés, elle peut-être déclarée `friend` .
2. Le paramètre `ostream` de `operator<<` doit être passé par référence pour effectuer la modification dans le flux en dehors de la fonction.
3. Il manque le `const` pour les méthodes `operato+`, et `operator *`

(c) **(3 points)** Implémentez, dans le fichier.cpp, les opérateurs suivants (tels que définis dans la définition de la classe), soit comme une méthode, soit comme une fonction globale (selon ce qui s'applique).

1. `operator+(MathVector)`,
2. `operator+=(MathVector)`,
3. `operator*(int, MatchVector)`, et
4. `operator*(MathVector)`

Solution:

```
MathVector MathVector::operator+(const MathVector& droite) {
    MathVector res;
    res.x_=x_+droite.x_;
    res.y_=y_+droite.y_;
    return res;
}

MathVector& MathVector::operator+=(const MathVector& droite) {
    *this=*this+droite;
    return *this;
}

int MathVector::operator*(const MathVector& droite) {
    return (x_*droite.x_)+(y_*droite.y_);
}

MathVector operator*(const int& gauche, const MathVector& droite) {
    MathVector res;
    res.x_=gauche*droite.x_;
    res.y_=gauche*droite.y_;
    return res;
}
```

Question 3 – Les Petits Mondes des Boules à Neige (5 points)

Gérard Tisan possède une petite fabrique de boules à neige. Il aime imaginer que ses boules à neige sont des petits mondes qui accueillent des créatures, et que ces mondes peuvent se dupliquer. Il a décidé de créer un programme pour le représenter, mais malheureusement, Gérard n'écoutait pas beaucoup en cours et a oublié des passages. Il vous présente donc le code suivant, avec des parties manquantes, en espérant que vous pourrez l'aider :

```
class Creature {
public:
    Creature(string nom = "Philippe"): nom_(nom) {}
    string getNom() const { return nom_; }
    void setNom(string nom) { nom_ = nom; }

private:
    string nom_;
};

class BouleANeige {
public:
    BouleANeige(int population_max = 10)
        : popmax_(population_max), pop_(0), habitants_(nullptr)
    {
        /* CODE MANQUANT 1 */
    }

    bool ajoutHabitant(Creature creature) {
        if (pop_ >= popmax_) {
            return false;
        }
        /* CODE MANQUANT 2 */
        return true;
    }

    void recenserPopulation() const {
        cout << "Population actuelle:" << endl;
        for (int i = 0; i < pop_; i++) {
            cout << " - " << i << ": " << habitants_[i].getNom() << endl;
        }
    }

private:
    int popmax_; /* Population maximale de la boule à neige */
    int pop_; /* Population actuelle de la boule à neige */
    Creature *habitants_; /* Habitants de la boule à neige */
};

int main () {
    BouleANeige boule1, boule2, copie1;

    boule1.ajoutHabitant(Creature("Lutin Tamarre"));
    boule1.ajoutHabitant(Creature("Lalie Corne"));
    boule1.ajoutHabitant(Creature("Fée Steve"));

    boule2.ajoutHabitant(Creature("Sucre d'Ogre"));
    boule2.ajoutHabitant(Creature("Edgar Gouille"));
    boule2.ajoutHabitant(Creature("Troll Lolo"));
```

```
    copie1 = boule1;
    copie1.recenserPopulation();

    BouleANeige copie2 = boule2;
    copie2.recenserPopulation();

    return 0;
}
```

On notera que les instructions sont écrites dans la définition de la classe.

(a) **(½ point)** Allouez la mémoire nécessaire pour qu'on puisse mettre des créatures dans la boule à neige en remplaçant le commentaire `/* CODE MANQUANT 1 */`.

Solution:

```
habitants_ = new Creature[popmax_];
```

(b) **(½ point)** Ajoutez les instructions nécessaires pour ajouter une créature dans les habitants de la boule à neige en remplaçant le commentaire `/* CODE MANQUANT 2 */`.

Solution:

```
habitants_[pop_++] = creature;
```

(c) **(1 point)** Gérard vous indique que lorsqu'il vérifie l'exécution du programme tel quel, il y a une fuite de mémoire à la fin de son exécution. Quelle est **la** méthode manquante pour éviter cette fuite de mémoire ? Donnez l'implémentation de cette méthode.

Solution:

Il s'agit du destructeur :

```
~BouleANeige() {
    delete [] habitants_;
}
```

(d) **(1 point)** Suite à votre correction du problème rencontré à la question précédente, Gérard vous dit que c'est pire : il y a maintenant une erreur de segmentation ! Expliquez brièvement pourquoi.

Solution:

À la question précédente, on désalloue la mémoire. Or, les objets `copie1` et `copie2` font des copies peu profondes (*shallow copy*) des objets `boule1` et `boule2`. Les attributs `habitants_` de l'objet original et de l'objet copié pointent alors vers le même espace mémoire. Lors de la fin d'exécution du programme, l'un de ces deux

objets est détruit en premier, menant à la désallocation de la mémoire pointée par les deux. Lors de la destruction du deuxième objet, on essaie de désallouer de la mémoire déjà désallouée : faute de segmentation.

(e) **(2 points)** En plus du code déjà ajouté aux questions précédentes, quelles sont les deux méthodes qu'il faudrait ajouter pour éliminer les fautes de segmentation ? Donnez l'implémentation de ces deux méthodes.

Solution:

Il s'agit du constructeur par copie :

```
BouleANeige(const BouleANeige &baneige)
    : popmax_(baneige.popmax_), pop_(baneige.pop_), habitants_(
       nullptr)
{
    habitants_ = new Creature[popmax_];

    for (int i = 0; i < pop_; i++) {
        habitants_[i] = baneige.habitants_[i];
    }
}
```

Et de l'opérateur d'affectation :

```
BouleANeige& operator=(const BouleANeige &baneige) {
    if (this != &baneige) {
        delete [] habitants_;

        popmax_ = baneige.popmax_;
        pop_ = baneige.pop_;

        habitants_ = new Creature[popmax_];
        for (int i = 0; i < pop_; i++) {
            habitants_[i] = baneige.habitants_[i];
        }
    }

    return *this;
}
```

Question 4 – Le problème du Cookie Monster (3 points)

Le Cookie Monster a décidé de se mettre à l'informatique après avoir entendu que les navigateurs Internet sont pleins de cookies. Pour son premier projet, il a choisi de faire de la programmation orientée objet, et de représenter son sac magique de friandises (majoritairement rempli de cookies... mais pas que).

— Dans le `SacMagique`, il y a plein de `Friandise` qui apparaissent de nulle part et y restent jusqu'à ce qu'elles soient mangées. On y trouve aussi des `Gnome` dont le rôle est de ranger les `Friandise`.

— Parmi les `Friandise`, on peut trouver des `Bonbon`, du `Chocolat`, du `Biscuit` et surtout des `Cookie`.
— Un `Cookie` est fait à partir de `Chocolat` et de `Biscuit`.
— Les `Gnome` doivent savoir quel est le `SacMagique` pour lequel ils font du rangement. Un `SacMagique` est rangé par plusieurs `Gnome`. Cependant, un `Gnome` ne travaille que dans un `SacMagique` et il doit savoir tout au long de son existence quel est le `SacMagique` dans lequel il va travailler.
— Pour ranger les `Friandise`, le `Gnome` en sélectionne une à la fois et la déplace au bon endroit. Pendant cette opération, la `Friandise` est toujours dans le `SacMagique`, et n'a jamais appartenu au `Gnome`. Un `Gnome` range des centaines voire des milliers de `Friandise` au cours de son existence.

Pour chacun des cas suivants, indiquez, à l'aide de la description ci-dessus, si les classes concernées ont une relation de composition, d'agrégation ou d'héritage. Dans le cas d'une relation d'agrégation, précisez si il faudrait utiliser une agrégation par pointeur ou par référence.

(a) **(½ point)** Quelle est la relation entre les classes `Cookie` et `Chocolat`, du point de vue de la classe `Cookie` ?

Solution:
Il s'agit d'une relation de composition.

(b) **(1 point)** Quelle est la relation entre les classes `SacMagique` et `Gnome`, du point de vue de la classe `Gnome` ?

Solution:
Il s'agit d'une relation d'agrégation par référence. En effet, le `SacMagique` n'appartient pas au `Gnome` puisqu'il peut être rangé par plusieurs `Gnome` (agrégation) mais un `Gnome` doit savoir tout au long de son existence quel est le `SacMagique` pour lequel il va travailler (référence).

(c) **(½ point)** Quelle est la relation entre les classes `Cookie` et `Friandise`, du point de vue de la classe `Cookie` ?

Solution:
Il s'agit d'une relation d'héritage.

(d) **(1 point)** Quelle est la relation entre les classes `Gnome` et `Friandise`, du point de vue de la classe `Gnome` ?

Solution:
Il s'agit d'une relation d'agrégation par pointeur. En effet, un `Gnome` déplace les `Friandise` mais elles ne lui appartiennent pas, et elle appartiennent toujours au `SacMagique`, il s'agit donc d'agrégation. Comme un `Gnome` range « des centaines voire des milliers » de `Friandise`, on sait que l'agrégation est par pointeurs.

Question 5 – Héritage (3 points)

Soit le code suivant :

```
class A {
public:
    void f() {
        cout << "A::f()" << endl;
    }
};

class B : public A {
public:
    void f() {
        cout << "B::f()" << endl;
    }
};

int main(void) {
    A a, *pa;
    B b, *pb;
    pa = &a;
    pb = &b;
    a.f(); pa->f();
    b.f(); pb->f();
    a = b;
    a.f();
    pa = pb;
    pa->f();

    return 0;
}
```

Si on exécute ce programme, quel sera l'affichage ?

Solution:

```
A::f()
A::f()
B::f()
B::f()
A::f()
A::f()
```