EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communication
Cours d'Informatique à la section SV
Sam J.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

INFORMATIQUE

Corrigé Examen Semestre II

Exercice 1 : Conception OO et programmation (48 points)

Il existe plusieurs façons de coder les détails des classes impliquées.
Notez que seuls les prototypes étaient demandés dans la partie 1.
Un corrigé possible est fourni dans les fichier sources `simulation.cc`

Exercice 2 : Questions sur les concepts (** points)

1. (a)
```
A()
B()
~B()
~A()
A()
B()
~B()
~A()
fin
```
justification: la ligne 19 appelle le constructeur par défaut de `B` qui appelle implicitement le constructeur par défaut celui de `A`. Le corps du constructeur de `B` est exécuté après celui de `A`. La ligne 20 appelle le destructeur de `B`. L'ordre d'appel des destructeurs est linverse de celui des consttructeurs. La ligne 22 provoque à nouveau l'appel au constructeur par défaut de `B`. Après la sortie du bloc se finissant à la ligne 23, le destructeur est appelé à nouveau pour le `B` déclaré dans le bloc interne (fin du bloc) puis le message `"fin"` s'affiche.

   (b) L'affichage devient:
```
A()
B()
A()
B()
~B()
~A()
~B()
~A()
fin
```

   (c) Il y a une fuite de mémoire car le `B` alloué dynamiquement n'est jamais désalloué

   (d) juste après la ligne 22: il y aura une double désallocation de mémoire, un objet statiquement alloué ne doit pas être désalloué explicitement. Juste après la ligne 23: le programme ne compile pas car la variable `b` est hors de portée.

2. (a) Le programme affiche :
```
1
1
5
```
justification : La ligne 32 déclare une variable `b` de type `A*` et l'initialise au moyen d'un pointeur sur un `B`. L'objet construit a un **attribut `a_` valant 1 et un attribut `b_` valant 2**. La ligne 33 invoque la méthode `A::m1`: **il n'y a pas résolution dynamique des liens car `m1` n'est pas virtuelle**. Cette méthode affiche 1. La ligne 34 cause l'appel à la méthode `B::m2` **car cette fois `m2` est virtuelle dans `A`** (polymorphisme). Cette méthode incrémente `b_` qui passe à 3, affiche `a_` par le biais de `A::m1` (ce qui affiche à nouveau 1) puis appelle la surcharge `B::m2(int)`. Cette méthode ajoute 2 à `b_` ce qui le fait passer à 5. Enfin `B::m2` affiche `b_`

   (b) Le programme affiche :
```
1
```
Il n'y a plus polymorphisme et c'est la méthode `A::m2` qui est appelée laquelle ne fait qu'incrémenter le `a_`.

   (c) Les méthodes suivantes ne peuvent pas être marquées `const` : `A:.m2()`, `B::m1()`, `B::m2()` et `B::m2(int)` car elles modifient directement un attribut ou font appel à une méthode les modifiant.

(d) Les méthodes suivantes ne peuvent pas être marquée `override` : `A::m1()`, `A::m2()`, `B::m1()` et `B::m2(int)` car elles ne redéfinissent pas une méthode virtuelle héritée de plus haut.

(e) l'instruction `ii`: la `i` n'est pas possible car `a_` est en accès privé dans `A`.

(f) non car il n'y a pas de méthode dans `A` ayant pour signature `m2(int)`

3. Il faut ajouter le code suivant à la classe `A`

```
public:
    virtual void affiche(ostream& out) const = 0;
```

et le code suivant en dehors de toute classe, après la classe `A`

```
ostream& operator<<(ostream& out, const A* a) {
    a->affiche(out);
    return out;
}
```

4. (a)
```
Guerrier(const Guerrier& other)
    : arme(new Arme(*other.arme)),
      bataillon(other.bataillon)
{}
```
Il faut faire de la copie profonde de l'arme car chaque `Guerrier` doit avoir sa propre arme. Il ne faut pas en faire pour le bataillon car le clone doit appartenir au même bataillon. Le constructeur de copie de `Arme` suffit car cette classe n'a pas de sous-classes.

(b)
```
Guerrier(const Guerrier& other):
    : arme(other.arme.copie()),
      bataillon(other.bataillon)
{}
```
Si `Arme` peut avoir des sous-classes alors la copie doit être polymorphique. Il faut programmer une méthode polymorphique `copie` dans la hiérarchie de `Arme`. Le constructeur de copie ne suffit plus car un constructeur n'est pas polymorphique.

Exercice 3 : Déroulement de programme [20 points]

Le programme affiche:

```
A() : 2
B() : 2
test:
1-> A::m1 : 4
2-> B::m1 : 3
A::m1 : 5
A::m3 : 6
B::m4 : 8
A::m2 : 8
B::m2 : 8
3-> A::m3 : 9
4-> B::m4 : 11
~B
~A
```

1. La ligne 68 construit un objet `b` de type `B` en appelant le constructeur de la ligne 36. Ce dernier fait appel au constructeur de la ligne 32 en lui passant 2 en argument. Ce constructeur fait appel au constructeur de la super-classe en lui passant 2 comme argument. L'unique attribut `a` de `B` (hérité de `A` prend alors la valeur 2 et le message :

   `A() : 2`

   La ligne 34 s'exécute ensuite ce qui affiche:

   `B() : 2`

   s'affiche (l'accès à `a` est possible car il est en protégé).

2. la ligne 70 appelle la fonction définie en ligne 57 en lui passant `b` en premier argument et 2 en second. L'affichage de la ligne 58 se produit, puis la méthode `m1(int)` est appelée sur le paramètre `a` qui se trouve être une **référence** à l'objet `b` défini en ligne 68. C'est la méthode `A::m1(int)` qui est appelée car **le paramètre `unB` est de type `A` et que `A::m1(int)` n'est pas virtuelle**. l'attribut `a` de `b` est incrémenté de la valeur de `val=2`(**grâce au passage par référence**) et le message :

   `1-> A::m1 : 4`

   s'affiche.

3. le message de la ligne 2 s'affiche. En ligne 61, c'est la méthode `B::m2()` qui est appelée sur `b`, **car `a` est un référence à `b` et que `m2` est virtuelle**.

4. en ligne 45 la méthode `m1(int, int)` de `B` est appelée (avec la valeur par défaut 2 du second argument). L'attribut de `a` prend la valeur $1 + 2 = 3$ et le message:

   `2-> B::m1 : 3`

   s'affiche.

5. en ligne 46 la méthode `A::m2` est appelée qui cause l'appel à `A::m1(2)` ce qui augmente `a` de 2 et affiche :

   `A::m1 : 5`

6. La ligne 15 d'exécute appelant `A::m3` (il y a virtualité mais `m3` **n'est pas redéfinie dans `B`**) ce qui augmente `a` de 1 et affiche:

   `A::m3 : 6`

   s'affiche.

7. la ligne 16 s'exécute ensuite, **la résolution dynamique des liens a lieu (`this` pointe sur un `B` et `m4` est virtuelle**. `a` est incrémenté de 2 et le message :

   ```
   B::m4 : 8
   ```

   s'affiche.
8. la ligne 17 s'exécute et le message

   ```
   A::m2 : 8
   ```

   s'affiche.
9. la ligne 47 s'exécute ensuite ce qui affiche:

   ```
   B::m2 : 8
   ```
10. l'exécution reprend en ligne 62 et la méthode `A::m3` est appelée (**elle est virtuelle mais pas redéfinie dans `B`**). Elle incrément `a` de 1 et affiche :

    ```
    3-> A::m3 : 9
    ```
11. La ligne 64 s'exécute puis il y a appel à `B::m4` (**appel via une référence et méthode virtuelle**). L'attribut `a` est augmenté de 2 et le message :

    ```
    4-> B::m4 : 11
    ```

    s'affiche
12. à la fin du `main` le destructeur est invoqué sur `b` (ordre inverse de celui de la construction) ce qui affiche :

    ```
    ~B
    ~A
    ```

Si l'on ajoute `private: int a =1;` L'affichage devient:

```
A() : 2
B() : 1
test:
1-> A::m1 : 4
2-> B::m1 : 3
A::m1 : 6
A::m3 : 7
B::m4 : 5
A::m2 : 7
B::m2 : 5
3-> A::m3 : 8
4-> B::m4 : 7
~B
~A
```