



TD2

Programmation sécurisée en C++ moderne

Date MAJ

17 nov. 2025

Exercice 1.

1) Améliorer et compléter le code ci-après en C++ moderne :

```
class A {
public:
    A() {
        x=new int ; *x=0 ;
        y=new int ; *y=0 ;
    }
    A(int _x,int _y) {
        x=new int ; *x=_x ;
        y=new int ; *y=_y ;
    }
    A(A&& o) {
        // Compléter le code
    }
    A& operator=(A&& o) {
        // Compléter le code
    }
    ~A() {
        if (x) delete x;
        if (y) delete y;
    }
private:
    int *x,*y;
    A(A&) {};
    A& operator=(const A&) {
    }
};
```

2) Modifier la classe précédente en ajoutant une méthode qui permet de renvoyer le nombre d'instances de la classe A.

3) Écrire une classe qui ne peut être instanciée qu'une seule fois

Exercice 2.

Réaliser une classe nommée `stack_int` permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un emplacement alloué dynamiquement ; sa dimension sera déterminée par l'argument fourni à son constructeur (on lui prévoira une valeur par défaut de 20). Cette classe devra comporter les opérateurs suivants (nous supposons que p est un objet de type `stack_int` et n un entier) :

- `<<`, tel que `p<<n` ajoute l'entier `n` à la pile `p` (si la pile est pleine, rien ne se passe) ;
- `>>`, tel que `p>>n` place dans `n` la valeur du haut de la pile `p`, en la supprimant de la pile (si la pile est vide, la valeur de `n` ne sera pas modifiée) ;
- `++`, tel que `p++` vale `true` si la pile `p` est pleine et `false` dans le cas contraire ;
- `--`, tel que `p--` vale `true` si la pile `p` est vide et `false` dans le cas contraire.

- On prévoira que les opérateurs `<<` et `>>` pourront être utilisés sous les formes suivantes (`n1`, `n2` et `n3` étant des entiers)

```
p << n1 << n2 << n3 ;
p >> n1 >> n2 << n3 ;
```

On fera en sorte qu'il soit possible de transmettre une pile par valeur. En revanche, l'affectation entre piles ne sera pas permise.

Exercice 3.

On veut concevoir un programme en C++17 qui modélise un train et ses diverses propriétés. On va donc concevoir une classe d'objets `Train`, qui pourrait par exemple être utilisée dans un programme simulant certains aspects du trafic ferroviaire.

Une instance de la classe `Train` est constituée de propriétés suivantes :

- Un train a un nom,
- Un train est composé d'un ensemble de wagons et d'une locomotive (une locomotive est considérée comme un wagon particulier), et chaque wagon est identifié par un numéro unique. Ce numéro est attribué automatiquement lors de l'acquisition (instanciation) du wagon.
- un train a une masse (incluant les wagons et leur contenu).

1) Soit la déclaration de classe `Wagon` :

```
struct Wagon {
    Wagon(const int masse);
    virtual int getMasse() const;
private:
    static int sCompteur;
    int mMasse;
    int mId;
};
```

Que faut-il faire pour ne pas permettre l'affectation d'une instance d'un `Wagon` dans une autre et éviter que deux instances possèdent le même identifiant ?

Implémenter les différentes méthodes de la classe.

2) La classe `Locomotive` est dérivée de la classe `Wagon`. Elle possède une propriété supplémentaire qui indique le **poids de son contenu**. La classe hérite des constructeurs de `Wagon` et possède un constructeur supplémentaire à deux paramètres : la **masse** du wagon et le **poids de son contenu**.

- Le poids de son contenu est initialisé à 0 s'il n'est pas précisé.
- Ce poids peut être modifié à travers la méthode `setPoidsContenu(const int p)`.

On donne la définition incomplète de la classe `Locomotive` :

```
struct Locomotive: public Wagon {
    Locomotive(const int masse,const int poids);
    void setPoidsContenu(const int p);
    int getMasse(); // Renvoie la somme du poids du contenu et la masse
private:
    int mMasseContenu;
};
```

Compléter la classe `Locomotive` et éventuellement la corriger (n'ajouter pas des nouvelles méthodes), puis définir ses méthodes.

3) On donne la définition de la classe `Train` suivante :

```

class Train {
public:
    Train(const std::string &nom);
    Train(const Train && autre);
    Train&& operator=(const Train && autre);
    void addWagon(Wagon *);
    void retirerWagon();
    int getMasse() const; // Renvoie la masse totale d'un train
private:
    std::string mName;
    std::vector<Wagon *> lWagons;
    Locomotive *mLocomotive {};
};

```

La méthode **addWagon()** permet d'ajouter un **Wagon** ou une **Locomotive**. Cette méthode doit vérifier s'il s'agit d'une locomotive. Si c'est le cas, l'insertion doit s'effectuer dans l'attribut **mLocomotive**. Sinon, lorsque l'on ajoute un wagon, l'insertion doit se faire dans le vecteur **lWagons**.

La méthode **retirerWagon()** permet de retirer le dernier wagon du vecteur **lWagons**. Si le vecteur est vide, la locomotive (représentée par **mLocomotive**) doit être retirée.

Indications : Utiliser les méthodes suivantes pour manipuler **std::vector<>**

- **push_back(const T& value)** : Ajoute un élément à la fin du vecteur.
- **pop_back()** : Supprime le dernier élément du vecteur.
- **size()** : Renvoie le nombre d'éléments dans le vecteur.

Exercice 4.

Écrire la classe Fraction pour permettre le code suivant :

```

#include <cassert>
auto main() -> int {
    Fraction f0 {5}; // f0=5/1
    assert(f0.valeur_reelle() == 5 && "5/1 équivaut à 5.");
    Fraction f1 { 5, 2 };
    assert(f1.valeur_reelle() == 2.5 && "5/2 équivaut à 2.5.");
    f1.simplifier();
    assert(f1.numerateur==5 && f1.denominateur==2 && "5/2 après
simplification donne 5/2.");
    f1 = pow(f1,2);
    assert(f1.numerateur == 25 && f1.denominateur == 4 && "5/2 au carré
équivaut à 25/4.");
    Fraction f2 { 258, 43 };
    assert(f2.valeur_reelle() == 6 && "258/43 équivaut à 6.");
    f2.simplifier();
    assert(f2.numerateur == 6 && f2.denominateur == 1 && "258/43 après
simplification donne 6/1.");
    f2 = pow(f2, 2);
    assert(f2.numerateur == 36 && f2.denominateur == 1 && "6/1 au carré
équivaut à 36/1.");
    return 0 ;
}

```

Soit la fonction **int foo(Fraction);**

Interdire l'appel à **foo(5);**