ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation Orientée Objet, section PH
Chappelier J.-C.

Programmation Orientée Objet (SPH) : CORRIGÉ DE L'Examen final

28 mai 2026

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre examen annulé dans le cas contraire.

1. Vous disposez d'une heure quarante-cinq minutes pour faire cet examen (9h15 – 11h00).

2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.

   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).

3. Vous avez droit à toute documentation papier.

   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.

4. Répondez aux questions directement sur la donnée. Ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.

   Si nécessaire, il y a une page supplémentaire en fin de copie.

5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, demandez des précisions à l'un(e) des assistant(e)s.

   Si un détail d'implémentation, un comportement ou une situation donnée n'est pas définie dans l'énoncé, vous êtes *libre* de le/la définir de la façon qui vous semble la plus adéquate. On considérera comme adéquate toute solution qui ne viole pas les contraintes données, ni les règles de bonne programmation, et qui ne résulte pas en un crash du programme.
   Cela veut aussi dire qu'il vous appartient de faire vos choix pour les parties non explicitement spécifiées (choisir les bons types, les bonnes d'implémentation, etc.).

6. L'examen comporte deux exercices indépendants, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 105) :
   1. quatre petites questions sur 34 points : $15 + 7 + 5 + 7$;
   2. sept analyses critiques de code sur 71 points : $14 + 12 + 10 + 10 + 5 + 6 + 14$.

   Tous les exercices comptent pour la note finale.

Question 1 – Cas d'école [34 points]

1.1 Un petit code [15 points]

Considérez le code ci-contre et répondez aux questions suivantes, en **justifiant** à chaque fois votre réponse :

① Si l'on supprimait le ligne `10`, le code compilerait-il ? (justifiez : si non, pourquoi ?)

② Le code fourni compile-t-il ? Si non, comment le corriger pour qu'il compile ?

③ En appliquant vos éventuelles corrections proposées en ②, qu'affiche-t-il une fois compilé et exécuté ?

④ A-t-il une (ou des) fuite(s) mémoire ?
Si oui : **(a)** lesquelles ? **(b)** comment les corriger ?

⑤ `D` a-t-elle un destructeur virtuel ?

Réponses et justifications :

① **Non** ça ne compilerait pas parce que l'appel ligne 49 ne peut pas se faire : nous somme dans la portée de `A` (puisque `el` est un `A*`) et que, sans la ligne 10, il n'y a pas de `f(string)` dans la portée de `A`.

Commentaire : Le fait que (si l'on supprime la ligne `10`) la classe `C` n'a pas de `f(string)` ne pose strictement aucun problème : il n'y a aucun appel à `f(string)` dans la portée de `C`.

② **Oui** le code fourni compile en l'état.

Commentaire : Beaucoup de mauvais arguments pour prétendre le contraire `:-(`
Il ne faut pas confondre mauviase pratique ou mauvaise implémentation (qui conduit à un mauvais comportement à l'*exécution*) avec erreur de **compilation**.

③ Il affiche :

```
D(3.14)
C::f([int] 9)
A::f("coucou")
C::f([int] 9)
A::f("coucou")
~C(1)
```

④ **Oui**, il y a une fuite mémoire.
(a) Le `double 3.14` pointé par `D` (`some_as[1]`).
(b) Cela vient du fait qu'`A` n'a pas de destructeur virtuel, donc la destruction des `A`s stockés dans `some_as` se fait « comme des `A` » et non pas comme les instances pointées ; le destructeur de `D`, qui fait le `delete` manquant, n'est donc pas appelé ;

pour corriger, il faut et il suffit d'ajouter un destructeur virtuel à `A` :

```
virtual ~A() = default;
```

Note : cela n'a rien à voir avec le destructeur virtuel de `D`. C'est bien du destructeur virtuel de `A` dont il s'agit ici.

⑤ **Oui**, `D` a un destructeur virtuel puisque qu'une de ses classes parentes (`C`) a un destructeur virtuel.

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class A {
public:
  virtual void f(int a) const = 0;

  void f(string const& s) { cout << "A::f(\"" << s << "\")" << endl;  }
};

class B : virtual public A {
public:
  void f(string const& s) { cout << "B::f(\"" << s << "\")" << endl; }
};

class C : virtual public A {
public:
  C(int c = 1) : my_c(c) {}
  virtual ~C() { cout << "~C(" << my_c << ')' << endl; }

  void f(int a) const override { cout << "C::f([int] " << a << ')' << endl; }

private:
  int my_c;
};

class D : public B, public C {
public:
  D(double u) : C(u), ptr(new double(u)) { cout << "D(" << u << ')' << endl; }

  ~D() { cout << "~D(" << *ptr << ')' << endl;
         delete ptr;                          }

private:
  double* ptr;
};

int main()
{
  C c1;

  vector<A*> some_as({ &c1 });
  some_as.push_back(new D(3.14));

  for (auto el : some_as) {
    el->f(9);
    el->f("coucou");
  }

  delete some_as.back();

  return 0;
}
```

1.2 Signaux [7 points]

① **[2 points]** Qu'affiche le programme ci-contre. **Justifiez** votre réponse.
Réponse et justification : Il affiche « `10 other samples` » (simple appel direct à `Another_Signal::print()`, la construction étant permise dans ce sens via le constructeur de `Another_Signal`).

② **[5 points]** Combien d'instances de `Another_Signal` sont crées en tout ?
Combien d'instances de `Signal` sont crées en tout ?
Combien de `double` toutes ces instances (`Signal` et `Another_Signal`) utilisent-elles en tout dans leurs attributs (c.-à-d. dans leurs `vector` ; on **ne** compte **pas** ici tous les autres `double` utilisés ailleurs) ?
Justifiez vos réponses.

Réponses et justifications : Il y a **1 instance** de `Another_Signal`, `s2`, et **2 ou 3 instances** de `Signal` : `s1`, la copie `x` de `s1` dans la construction de `s2` et, ou non suivant comment on comprend la question, `s2` lui-même puisque `Another_Signal` **est un** `Signal`.

Tout ceci crée **40** `double` : quatre fois 10 : 10 pour `s1`, 10 pour la copie de `s1` passé au constructeur de `s2` (le `x` passé par valeur), et `s2` lui-même en contient deux (chacun de 10) : un direct et un par héritage.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

double f1(double t) { return 4.1 * sin(37.5 * t + 0.5); }
double f2(double t) { return 2.8 * sin(54.6 * t + 1.9); }

class Signal {
public:
  Signal(size_t size, double T_e = 1.0) : content(size)
  { for(size_t n(0); n < size; ++n) { content[n] = f1(n * T_e); } }

  virtual void print() const
  { cout << size() << " samples" << endl; }

  size_t size() const { return content.size(); }

private:
  vector<double> content;
};

class Another_Signal : public Signal {
public:
  Another_Signal(Signal x, double T_e = 1.0) : Signal(x), more_content(x.size())
  { for(size_t n(0); n < more_content.size(); ++n)
    { more_content[n] = f1(n * T_e) + f2(n * T_e); }
  }

  virtual void print() const override
  { cout << more_content.size() << " other samples" << endl; }

private:
  vector<double> more_content;
};

int main()
{
  Signal s1(10);
  Another_Signal s2(s1);
  s2.print();
  return 0;
}
```

1.3 Notes [5 points]

Un enseignant souhaite pouvoir stocker dans un programme C++ la note obtenue pour chacun de ses étudiant(e)s au moyen de leur SCIPER. Par exemple si l'étudiant(e) de SCIPER 403968 a eu un 5.75, il voudrait faire quelque chose comme : `notes.ajoute(403968, 5.75);`.
Il veut aussi pouvoir retrouver facilement la note associée à un SCIPER donné.

(Remarque : si la note existe déjà, `ajoute()` la remplace simplement par la nouvelle note.)

Cet enseignant a 430 étudiant(e)s dont les SCIPER vont de 264996 à 403968.

Définissez (minimalement) la classe `Notes` pour stocker ses notes : donnez le ou les attributs et la définition complète de la méthode `ajoute()`.

Justifiez vos choix d'implémentation.

Réponse et justifications : Vus les valeurs envisagées, il n'est pas raisonnable de stocker tous les SCIPER entre 264996 et 403968. L'idée est donc d'avoir *uniquement* les 430 SCIPER utiles. Le plus simple pour cela est d'utiliser une table de hashage (`map`) :

```
class Notes {
public:
  void ajoute(unsigned int SCIPER, double grade) {
    content[SCIPER] = grade;
  }
private:
  map<unsigned int, double> content;
};
```

On pourrait aussi imaginer faire soi-même l'association, p.ex. avec une classe (ou `struct`) de paires (SCIPER, note) que l'on pourrait alors stocker dans un tableau, trié par SCIPER pour permettre une recherche par dichotomie.

Notes :

- Avec cette dernière solution, utiliser un `vector` non trié ou un `set` serait possible mais sous-optimal du point de vue théorie, puisqu'on ne saurait pas retrouver une note à partir d'un SCIPER autrement que par itération sur tous les éléments du `set`. [1]
  De plus, mais ce n'est pas demandé ici, un `set` ne permettrait pas de stocker plusieurs notes pour un même SCIPER (à moins qu'elles soient systématiquement toutes différentes...)
- Il n'y a rien de particulier avec les SCIPER donnés (autre que la plage d'entiers utilisée est très vide) et il n'est donc pas nécessaire de stocker ces valeurs min et max.
  Si l'on voulait vraiment ajouter une fonctionnalité comme tester si un SCIPER est correct pour la classe, alors il faudrait avoir la liste explicite des 403 SCIPER de cette classe.

Commentaire : Certain(e)s ont imaginé une « fonction fictive » qui ferait une injection des 403 SCIPERs dans $[\![1, 403]\!]$. Mais il n'est pas possible de construire une telle fonction de façon générique (autrement qu'en la définissant point par point... ...avec une `map`, par exemple). Ceci contourne donc totalement la question.

1. J'aurais dû aussi demander la méthode `double note(unsigned int sciper);`...

1.4 Employé(e)s [7 points]

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Person {
public:
  Person(const string& name) : name_(name) {}
  virtual ~Person() = default;
  virtual void salary() = 0;
private:
  const string name_;
};

class Company {
public:
  Company() {}
  void hire(Person* p)
  { if (p != nullptr) employees.push_back(p); }
private:
  vector<Person*> employees;
  Company(Company const&)             = delete;
  Company& operator=(Company const&) = delete;
};

class Worker : public Person {
public:
  void salary() override
  { cout << "worker salary" << endl; }
};

int main()
{
  Worker w("Eddy");
  Company c;
  // ...à vous de continuer...

  return 0;
}
```

Un étudiant a commencé le code ci-contre et aimerait ajouter le `Worker w` à la `Company c`.

① Le code écrit jusque là compile-t-il ? Si non expliquez pourquoi et proposez une correction.

② Une fois le code corrigé si nécessaire, est-il possible ajouter `w` à `c` ? Si oui écrivez ici le code pour le faire, sinon expliquez pourquoi ce n'est pas possible.

Réponses :

① **Non**, parce que `Worker` n'a pas de constructeur prenant une `string` (on n'hérite pas les constructeurs ; elle ne possède donc que le constructeur de copie par défaut (et celui de déplacement). À noter que le constructeur par défaut par défaut n'existe pas non plus pour `Worker` puisque `Person` n'a pas de constructeur par défaut).

Le plus simple pour corriger cela est de remettre le constructeur de `Person` dans la portée de `Worker` (« héritage » de constructeur) :

```
using Person::Person;
```

On peut, bien sûr, aussi le faire autrement en écrivant plus de code.

② **Oui**, pas de problème : un `Worker` est une `Person` (héritage). Il suffit donc d'écrire :

```
c.hire(&w);
```

Question 2 – Diverses conceptions [71 points]

Dans un cours de programmation C++, un projet a été proposé pour coder un système de simulation d'interaction entre particules [2].

Nous allons suivre la conception, et les erreurs, de différentes versions. Votre but sera essentiellement de proposer des corrections ou compléter des classes, mais chaque sous-question décrit plus en détails ce qui est attendu à chaque fois.

2.1 Un mauvais départ [14 points]

Un groupe a commencé son projet en écrivant le code suivant (après les `#include` nécessaires) :

```
class Vector3D {
private:
  double x, y, z;

public:
  Vector3D(double a = 0.0, double b = 0.0, double c = 0.0) : x(a), y(b), z(c) {}

  double norme2() const { return double(x*x + y*y + z*z); }

  double norme() const { return double(sqrt(x*x + y*y + z*z)); }

  // vecteur unitaire colineaire et de meme sens
  Vector3D& operator~() const
  { return Vector3D(x / norme(), y / norme(), z / norme()); }

  bool operator==(const Vector3D& autre) const
  { return ((x == autre.x) and (y == autre.y) and (z == autre.z)); }
};
```

qui provoque cette erreur de compilation :

```
vector3d.cc: In member function 'Vector3D& Vector3D::operator~() const':
vector3d.cc: error: cannot bind non-const lvalue reference of type 'Vector3D&' to an rvalue
                    of type 'Vector3D'
      |   { return Vector3D(x / norme(), y / norme(), z / norme()); }
      |              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

① **[2 points]** Corrigez (sur le code) l'erreur pour que le code compile.

Supprimer la référence sur le type de retour de `operator~()` : simplement `Vector3D`.

② **[6 points]** Quel(s) autre(s) conseil(s) pourriez-vous donner à ce groupe ?
Essayez d'être exhaustive/exhaustif.

Réponses :

- Ne pas comparer des `double` avec `==`.
- Pas de copié-collé (`norme()` et `norme2()`).
- Protéger `operator~()` contre la division par 0 (vecteur nul).
- Ne pas oublier `operator!=()` si l'on compile avec des anciennes normes (depuis C++20 cet opérateur est automatiquement fourni par le compilateur si `operator==()` a été défini).
- Continuer la classe, en particulier l'aspect algébrique des vecteurs.

  (et, typiquement, réécrire `operator~()` en utilisant la multiplication d'un vecteur par un scalaire...)
- (mineur, style) Il est inutile de mettre les casting `double(...)` dans les expression de retour des normes.

2. Toute ressemblance avec un projet existant ne serait pas fortuite.

Plus loin dans leur projet vous voyez du code ressemblant à ceci :

```
vector<double> speed = compute_speed(some_particle);

if (sqrt(speed[0]*speed[0] + speed[1]*speed[1] + speed[2]*speed[2]) >= some_speed) {
  do_something(some_particle);
} else {
  do_something_else(some_particle);
}
```

③ **[6 points]** Quel(s) autres conseil(s) voudriez-vous donner à ce groupe ?
Réécrivez (ensuite, *après* les conseils) le code C++ que vous proposeriez *pour cette partie* (en supposant écrit tout le reste que vous souhaiteriez, mais sans nécessairement l'expliciter ici).

Réponse :

— Pourquoi utiliser `vector` plutôt que `array` alors que la taille est connue *a priori* et ne change pas ?

Encore bien mieux : utiliser la classe `Vector3D`

— Utiliser `Vector3D::norme()` (ou au moins écrire une fonction à partager ; pas de « copié-collé »)

— Au final, on pourrait même carrément écrire simplement une méthode pour `some_particle`, vu que tout ne dépend que de cette variable (et `some_particle` utiliserait en interne des `Vector3D`).

Tout ce code deviendrait alors simplement :

```
some_particle.do_something_depending_on_speed();
```

2.2 Une première version de particules [12 points]

Voici ci-contre un code résumé d'une première ébauche de tentative, laquelle comprend plusieurs erreurs, à commencer (mais ce ne sont pas les seules) par des erreurs de compilation :

```
projet.cc:8:8: error: initializer specified for non-virtual method
                      'void Printable::print(std::ostream&) const'
    8 |   void print(ostream& out) const = 0;
      |        ^~~~~
projet.cc: In function 'std::ostream& operator<<(std::ostream&, const Printable&)':
projet.cc:14:1: warning: no return statement in function returning non-void [-Wreturn-type]
   14 | }
      | ^
projet.cc: At global scope:
projet.cc:23:8: error: 'void Flake::print(std::ostream&) const' marked 'override', but does
                      not override
   23 |   void print(ostream& out) const override { out << "un flocon"; }
      |        ^~~~~
projet.cc:28:8: error: 'void Stone::print(std::ostream&) const' marked 'override', but does
                      not override
   28 |   void print(ostream& out) const override { out << "un caillou"; }
      |        ^~~~~
projet.cc:39:8: error: 'void System::print(std::ostream&) const' marked 'override', but does
                      not override
   39 |   void print(ostream& out) const override {
      |        ^~~~~
projet.cc: In member function 'void System::print(std::ostream&) const':
projet.cc:40:20: error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::unique_ptr(const
                       std::unique_ptr<_Tp, _Dp>&) [with _Tp = Particle; _Dp =
                       std::default_delete<Particle>]'
   40 |     for (auto el : content) { out << "- " << el << endl; }
      |                    ^~~~~~~
In file included from /usr/include/c++/15/memory:80,
                 from projet.cc:3:
/usr/include/c++/15/bits/unique_ptr.h:524:7: note: declared here
  524 |       unique_ptr(const unique_ptr&) = delete;
      |       ^~~~~~~~~~
projet.cc:40:20: note: use '-fdiagnostics-all-candidates' to display considered candidates
   40 |     for (auto el : content) { out << "- " << el << endl; }
      |                    ^~~~~~~
projet.cc:40:43: error: no match for 'operator<<' (operand types are
                      'std::basic_ostream<char>' and 'std::unique_ptr<Particle>')
   40 |     for (auto el : content) { out << "- " << el << endl; }
      |                                      ~~ ~~
```

```cpp
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

class Printable {
public:
  void print(ostream& out) const = 0;
};

ostream& operator<<(ostream& out, Printable const& obj)
{
  obj.print(out);
}

class Particle : public Printable {
  // Il y a plus de code ici, mais peu importe pour le moment.
  // Cependant, Particle est une classe abstraite.
};

class Flake : public Particle {
public:
  void print(ostream& out) const override { out << "un flocon"; }
};

class Stone : public Particle {
public:
  void print(ostream& out) const override { out << "un caillou"; }
};

class System : public Printable {
private:
  vector<unique_ptr<Particle>> content;

public:
  System(vector<Particle*> const& list)
  { for (auto ptr : list) content.push_back(unique_ptr<Particle>(ptr)); }

  void print(ostream& out) const override {
    for (auto el : content) { out << "- " << el << endl; }
  }
};

int main()
{
  Stone c;
  Flake f;
  System sys({ &c, &f });

  cout << sys << endl;

  return 0;
}
```

Corriger **toutes** les erreurs du code précédent (celles de compilation et celle(s) de conception). Apportez **directement vos corrections** sur le code donné précédemment, avec une *brève* explication à coté, et **indiquez ci-dessous** quelles lignes vous avez corrigées avec, *si nécessaire*, une explication complémentaire plus détaillée.

Note : on **ne** s'intéresse **pas** au style ni aux instructions possiblement inutiles.

Lignes corrigées : Il y a erreurs de compilation et 1 erreur de conception :

- ligne 8, il manque le mot clé `virtual` ;
- ligne 13–14, il manque le `return out;` ;

  la correction consistant à changer le type de retour par `void` n'est pas valide en raison de la ligne 50 ;
- ligne 40, il manque le `const&` : le parcourt des `unique_ptr` doit se faire sans copie (par référence, donc) ;
- ligne 40, il manque l'étoile (`*`) devant l'accès à `el` : on n'affiche pas le pointeur, mais la valeur pointée.
- L'erreur de conception est de mettre des adresses allouées statiquement (ligne 48) dans des `unique_ptr` (ligne 37) ce qui va provoquer une erreur (« segmentation fault ») lors de leur libération.

  On peut corriger cette erreur, soit en changeant les `unique_ptr` pour des pointeurs à la C (le plus simple), soit en allouant dynamiquement une copie de `c` et `f`.

  Dans la version « pointeur à la C », la première erreur de la ligne 40 n'en est alors plus une (on peut laisser sans le `const&`). Je l'ai néanmoins laissée dans le corrigé ci-dessous pour celles et ceux qui ne proposent pas le changement en « pointeur à la C ».

Note : bien que pas strictement nécessaire dans cet exemple, il est bon d'ajouter un destructeur virtuel à `Printable`.

Voici une correction possible, la plus simple :

```cpp
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

class Printable {
public:
  virtual void print(ostream& out) const = 0;
};

ostream& operator<<(ostream& out, Printable const& obj)
{
  obj.print(out);
  return out;
}

class Particle : public Printable {
  // Il y a plus de code ici, mais peu importe pour le moment.
  // Cependant, Particle est une classe abstraite.
};

class Flake : public Particle {
public:
  void print(ostream& out) const override { out << "un flocon"; }
};

class Stone : public Particle {
public:
  void print(ostream& out) const override { out << "un caillou"; }
};

class System : public Printable {
private:
  vector<Particle*> content;

public:
  System(vector<Particle*> const& list)
  { for (auto ptr : list) content.push_back(ptr); }

  void print(ostream& out) const override {
    for (auto const& el : content) { out << "- " << *el << endl; }
  }
};

int main()
{
  Stone c;
  Flake f;
  System sys({ &c, &f });

  cout << sys << endl;

  return 0;
}
```

2.3 Autre tentative [10 points]

L'un des deux membres du groupe décide alors de partir sur une autre approche, avec sa propre allocation mémoire. Il a par ailleurs travaillé sur le graphisme et utilise un `raylibRender` pour visualiser leur projet (peu importe les détails ici).

Voici ci-contre les parties pertinentes de leur code pour cette question. Tout le reste est dans le même esprit que la sous-question précédente et tout le code compile parfaitement sans erreur. Le programme se lance également sans erreur et visualise correctement ce qu'ils attendent. Cependant, à chaque fois qu'ils terminent le programme, celui-ci plante sur une erreur système qu'ils ne comprennent pas : `Segmentation fault (core dumped)`

```
1  // ... d'autres choses avant...
2
3  class System : public Printable {
4  private:
5    vector<Particle*> content;
6
7  public:
8    System(vector<Particle*> const& list)
9    { for (auto ptr : list) content.push_back(ptr); }
10
11   ~System()
12   { for (auto ptr : content) delete ptr; }
13
14   // règle des 3:
15   System(System const&)            = default;
16   System& operator=(System const&) = default;
17
18   void print(ostream& out) const override;
19 };
20
21 class raylibRender : public SupportADessin {
22 public:
23   raylibRender(System const& sys) : s(sys) {}
24
25   void run() { cout << s; } // pour simplifier ici
26
27   // ...plein d'autres choses non pertinentes ici...
28
29 private:
30   System s;
31   // ...plein d'autres choses non pertinentes ici...
32 };
33
34 // ...d'autres choses non pertinentes ici aussi...
35
36 int main()
37 {
38   System sys({ new Stone, new Flake });
39
40   raylibRender screen(sys);
41   screen.run();
42
43   return 0;
44 }
```

Expliquez le problème et proposez une correction *complète* (indiquez clairement toutes les lignes de vous corrigez et pourquoi).

Réponse : (vous pouvez aussi répondre ci-dessus)

Le problème vient de l'incohérence de conception dans la gestion du contenu du `System`. Plus précisément, si l'on met un `delete` dans le destructeur, alors c'est que l'on décide que `System` gère son contenu pointé et donc on doit implémenter une copie profonde (règle des trois) et non pas garder la copie de surface par défaut.

Cette erreur de conception se traduit dans la copie du `System s` dans le constructeur de `raylibRender` (ligne 23).
En raison de cette copie, *deux* `System` sont détruits qui pointent vers le même contenu et donc il y a *deux* fois le `delete` de ce contenu.

La correction la plus simple, qui est de toutes façons bonne à faire mais qui ne corrige pas **le fond** du problème, est de mettre une référence vers un `System` en ligne 30.
(Note mineure : il faudrait alors enlever le `const` de l'argument passé au constructeur ligne 23. Mais ceci n'est pas attendu dans un examen sur papier (sans compilateur).)
On peut aussi (mais un peu moins bien à mon avis ici) utiliser un pointeur à la C (mais en **aucun cas** un smart-pointer !).

La vraie correction **de fond** est de ne pas garder la copie de surface du contenu pointé par le `System`. Le plus simple est de simplement mettre `delete` à la place de `default` en lignes 15 et 16 pour empêcher toute copie de `System`.

On pourrait aussi explicitement coder la copie profonde, mais cela nécessite de supposer la copie de `Particule`.

On peut aussi décider de ne pas transférer la propriété des `Particule` au `System`, mais de la laisser à l'extérieur :

- supprimer les lignes 11 à 17
- soit surprimer les `new` ligne 38 et créer deux variables (alouées statiquement ; passer leur adresse) ; soit sauvegarder ces deux pointeurs dans des variables et les `delete` en fin de `main()`

Mais cette approche n'est conceptuellement pas bonne car :

- elle ne respecte pas le « *partir sur une autre approche* » de départ (elle revient à l'approche de la question précédente) ;
- si on ne supprime pas la copie du `System`, on aura plusieurs `System` qui vont vouloir modifier « leurs » particules (lesquelles sont partagées) !

Choisir cette solution montre donc qu'on n'a pas vraiment compris le problème de fond.

On peut enfin décider de revoir toute la conception et y mettre des `unique_ptr`, mais ceci me semble trop de travail ici.

Commentaire : Les réponses à ces questions 2.2 et 2.3 montrent que, malheureusement, les questions de gestion mémoire, de « propriétaire », etc., ne sont pas maîtrisées par la plupart.

2.4 Ils avancent [10 points]

Ayant réussi à l'aide d'une assistante à corriger le bug précédent, ils avancent maintenant sur la simulation des interactions.
Au niveau de leur classe abstraite `Particle`, ils ont trois méthodes `addForce()` différentes :

```
class Particle : public Printable {
public:
  void addForce(Force const& new_f) { cout << "add force" << endl; }
  void addForce() { cout << "default add force: add g" << endl; }
  virtual void addForce(Particle const& other) = 0;
  // ...plein d'autres choses non pertinentes ici...
};
```

et ils redéfinissent la troisième dans les sous-classes :

```
class Flake : public Particle {
public:
  void addForce(Particle const& other) override { cout << "interaction with flakes" << endl; }
  // ...plein d'autres choses non pertinentes ici...
};

class Stone : public Particle {
public:
  void addForce(Particle const& other) override { cout << "interaction with stones" << endl; }
  // ...plein d'autres choses non pertinentes ici...
};
```

Ils font le petit test suivant (avec les `#include` nécessaires) :

```
 6 int main()
 7 {
 8   Stone c;
 9   Flake f;
10   const Force force(0.0, 0.0, -4e-5);
11
12   c.addForce(f);
13   f.addForce(f);
14   c.addForce(c);
15   f.addForce(c);
16
17   c.addForce(force);
18
19   return 0;
20 }
```

Mais celui-ci ne compile pas. Ils ont cette erreur :

```
test_addForce.cc: In function ‘int main()’:
test_addForce.cc:17:14: error: cannot convert ‘Force’ to ‘const Particle&’
   17 |   c.addForce(force);
      |              ^~~~~
      |              |
      |              Force
stone.h:21:33: note: initializing argument 1 of ‘virtual void Stone::addForce(const Particle&)’
   21 |   void addForce(Particle const& other) override { cout << "interaction with ston [...]
      |                 ~~~~~~~~~~~~~~~~~~~~^~~~~
```

Expliquez quelle est la source de l'erreur et comment ils doivent la corriger.

Réponse : (vous pouvez aussi répondre sur la page précédente)

On a ici un problème de masquage et de résolution de portée : `Stone::addForce(Particle const&)` *masque* tous les `Particle::addForce(...)`.

Or l'appel ligne 17 se fait dans la portée de `Stone`. On n'y « voit » donc plus les autres `Particle::addForce(...)`.

La solution consiste à démasquer lors de l'appel :
```
c.Particle::addForce(force);
```

On pourrait aussi changer la portée dans laquelle se fait l'appel :
```
Particle* ptr = &c;
ptr->addForce(force);
```
Il faut ici impérativement un `Particule*` et non pas un `Stone*` qui resterait dans la même portée.

(Et cette question n'a rien à voir avec le polymorphisme ou la résolution dynamique des liens.)

Redéfinir les méthodes dans les sous-classes serait aussi une solution, mais mauvaise car :

1. rien de nouveau n'est fait, c'est donc du travail *inutile* ;
2. cela obligerait à le refaire dans chaque nouvelle sous-classe, donc inutile (point précédent) **et** *fastidieux*.

Ajouter `using Particle::addForce;` est une variante plus légère de la même idée, sans redéfinition, mais impose quand même de le faire dans chaque sous-classe. Ca me semble moins bien que de désambiguiser l'appel car c'est la responsabilité de l'appelant de gérer son problème sans imposer des contraintes à ceux/celles qui écrivent les sous-classes (après, ça dépend respectivement du nombre de sous-classes et du nombre de tels appels).

Commentaire : Trop de confusions entre redéfinition (polymorphisme) et masquage.

2.5 Ajout par surcharge de << [5 points]

Un autre groupe à conçu son `System` comme ceci :

```
class System : public Printable {
public:
  void add(Particle const& particle);
  // ...plein d'autres choses non pertinentes ici...
private:
  vector<unique_ptr<Particle>> content;
};
```

avec une copie polymorphique des particules :

```
class Particle : public Printable {
public:
  virtual unique_ptr<Particle> copy() const = 0;
  // ...plein d'autres choses non pertinentes ici...
};
```

Elles veulent également avoir l'ajout de `Particle` à leur `System` au moyen de l'opérateur `<<` pour pouvoir par exemple faire :

```
systeme << flocon1 << flocon2 << caillou1;
```

Définissez ici l'opérateur `<<` correspondant et la méthode `add()` :

Réponse :

```
void System::add(Particle const& particle)
{
  content.push_back(particle.copy());
}

System& operator<<(System& sys, Particle const& particle)
{
  sys.add(particle);
  return sys;
}
```

On peut aussi faire une surcharge interne si on préfère :

```
System& operator<<(Particle const& particle)
{
  add(particle);
  return *this;
}
```

Note : `Particle` étant une classe *abstraite* (elle a au moins la méthode virtuelle pure `copy()`), on **ne** peut **pas** en faire de copie directe (donc des choses du genre « `make_unique<Particle>(particle)` » ne sont pas possibles).

Commentaire : Pourquoi ne pas donner la définition de la méthode `add()` alors que c'est explicitement demandé ?

2.6 Sources de particules [6 points]

Ce même groupe souhaite faire des sources de particules (abstraites, donc) qui génère des variations aléatoires d'une particule de référence stockée dans la source.

Elles ont pour cela la classe `Source` suivante :

```
class Source {
public:
  Source(Particle const& example) : reference(example) {}
  unique_ptr<Particle> generate() const;

private:
  Parameters randomize() const;

  Particle const& reference;
};
```

dans laquelle `Source::randomize()` génère aléatoirement un nouveau jeu de paramètres.

Par ailleurs, leur classe `Particle` a une méthode

```
void Particle::update(Parameters const& new_parameters);
```

qui permet de modifier l'instance courante en fonction des nouveaux paramètres (p.ex. ces paramètres contiennent une nouvelle position et une nouvelle vitesse).

Elles souhaitent que la méthode `Source::generate()` génère une nouvelle version aléatoire de la particule de référence de la source, en utilisant une version `randomize()` de ses paramètres.

Définissez ici la méthode `Source::generate()` :

```
unique_ptr<Particle> Source::generate() const
{
  unique_ptr<Particle> retour(reference.copy());
  retour->update(randomize());
  return retour;
}
```

Note : `reference` étant une référence vers une classe *abstraite*, on **ne** peut **pas** en faire de copie directe (donc des choses du genre « `make_unique<...>(reference)` » ou « `Particle p(reference);` » ne sont pas possibles).

2.7 Plus d'obstacles [14 points]

Un troisième groupe a défini une classe (concrète) `Obstacle` :

```
class Obstacle : public Printable {
public:
  void interact_with(Particle& particle) {
    // pour simplifier ici :
    print(cout); cout << " agit sur "; particle.print(cout); cout << endl;
  }
  void print(ostream& out) const override { out << "un obstable"; }
};
```

et a mis dans son `System` un `vector<Obstacle>`. Tout compile et fonctionne correctement, mais ils souhaitent maintenant avoir plusieurs type d'obstacles différents.

Plus concrètement, ils veulent :

- garder la classe (concrète) existante, quitte à en changer de nom, mais avoir plusieurs autres sortes d'obstacles possibles, chaque obstacle ayant au moins les méthodes `print()` et `interact_with()`, à chaque fois avec des comportements spécifiques ;
- pouvoir avoir n'importe quel type d'obstacles dans la collection stockée dans leur `System` ;
- avoir un autre obstacle concret, `Table`, qui réalise un choc élastique de la `Particle` (expliqué plus loin).

2.7.1 Comment doivent-ils procéder ? [6.5 points]

Comment doivent-ils procéder ?
Décrivez ici les principales étapes de la transformation de leur code.
Une description claire en français (ou anglais) peut suffire ; vous pouvez si nécessaire ajouter du code C++, mais ce n'est pas obligatoire ici.

Réponse : Il faut :

- renommer la classe `Obstacle` existante (p.ex. en `ObstacleType1`) et la faire hériter de la classe `Obstacle` ;
- créer la super-classe `Obstacle` et y mettre les aspects généraux, en tout cas la méthode `interact_with()` qui devient virtuelle pure ;
- créer la classe `Table` qui hérite d'`Obstacle` ;
- transformer la collection d'obstacle du `System` en une collection de *pointeurs* sur des obstacles (type de pointeur au choix suivant le modèle de gestion mémoire choisi) ;
- et, bien sûr, reporter cette adaptation à toutes les méthodes de `System` qui le nécessitent (p.ex. le constructeur).

Voici, page suivante, une solution possible.

```
class Obstacle : public Printable {
public:
  virtual void interact_with(Particle& particle) = 0;
};

class  ObstacleType1 : public Obstacle {
public:
  virtual void interact_with(Particle& particle) override {
    // pour simplifier ici :
    print(cout); cout << " agit sur "; particle.print(cout); cout << endl;
  }
  void print(ostream& out) const override { out << "un obstable"; }
};

class Table : public Obstacle; // etc. comme dans la donnée

class System : public Printable {
private:
  vector<Obstacle*> content;

public:
  System(vector<Obstacle*> const& list)
  { for (auto ptr : list) content.push_back(ptr); }
  // ...etc.
};
```

Note : le destructeur d'`Obstacle` serait virtuel par héritage de `Printable`.

2.7.2 Les tables [7.5 points]

Ils souhaitent que leur obstacle concret `Table` soit défini par un vecteur normal (ils le considèrent comme un plan infini) et que sa méthode `interact_with()` réalise un choc élastique de la `Particle` sur ce plan.

Définissez ci-dessous complètement, *sauf* sa méthode `print()`, la classe `Table`, en sachant que dans leur projet ils ont :

- une classe `Vector3D` avec les opérateurs usuels, dont l'opposé avec `-` ;
- une *fonction*

  ```
  Vector3D symetry(Vector3D const& axis, Vector3D const& input);
  ```

  qui retourne le vecteur symétrique de `input` par rapport à `axis` ;
- les méthodes

  ```
  Vector3D Particle::get_speed()                  const ;
  void     Particle::set_speed(Vector3D const& s) ;
  ```
- qu'un choc élastique consiste simplement à transformer la vitesse en l'opposé de son symétrique par rapport à la normale de l'obstacle au point de contact.

Réponse :

```
class Table : public Obstacle {
public:
  Table(Vector3D n) : normal(n) {}

  virtual void interact_with(Particle& particle) override {
    particle.set_speed(- symetry(normal, particle.get_speed()));
  }

private:
  Vector3D normal;
};
```