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

22 mai 2025

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**.
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, ou si vous avez un doute, demandez des précisions à l'un(e) des assistant(e)s.
6. L'examen comporte quatre 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 110) :
   1. deux petites questions : 10 points ;
   2. conception et programmation 1 : 40 points ;
   3. correction de code : 20 points ;
   4. conception et programmation 2 : 40 points.

   Tous les exercices comptent pour la note finale.

Question 1 – Faisons le point [10 points]

1.1 Trois petits codes [5 points]

Pour chacun des trois codes proposés ci-dessous, dites :

1. s'il compile ;
2. si oui, s'il s'exécute correctement ; (et sinon laissez blanc ;)
3. si oui aux deux questions précédentes : ce qu'il affiche ;
   et si non à l'une des deux questions précédentes : la cause de l'erreur ;
4. s'il y a une fuite de mémoire ; et si oui : sur qui ?/pourquoi ?

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

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

int main()
{
  double a(12.34);
  double* ptr(&a);
  cout << *ptr << endl;
  ptr = nullptr;
  return 0;
}
```

code1.cc

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

class A {
public:
  A(double x)
  : ptr(new double(x))
  {}
  ~A() { ptr = nullptr; }
  void print() const
  { cout << *ptr << endl; }
private:
  double* ptr;
};

int main()
{
  A a(5.67);
  a.print();
  return 0;
}
```

code2.cc

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

class A {
public:
  A(double x)
  : ptr(make_unique<double>(x))
  {}
  void print() const
  { cout << *ptr << endl; }
private:
  unique_ptr<double> ptr;
};

int main()
{
  A a(89.1);
  a.print();
  return 0;
}
```

code3.cc

	code1.cc	code2.cc	code3.cc
compile ? (oui/non)	oui	oui	oui
s'exécute ? (oui/non)	oui	oui	oui
affichage ou cause de l'erreur	12.34	5.67	89.1
fuite mémoire ? (oui/non) + brève explication	non : toute la mémoire est allouée statiquement	oui : le `new` du constructeur n'a pas de `delete`	non : le `unique_ptr` libère lui-même la mémoire allouée

Commentaires : Surtout aucun `delete` dans le `code1.cc` : on est dans un cas d'utilisation *de références*, pas celui d'allocation dynamique !
Par ailleurs, une fuite de mémoire (`code2.cc`) n'empêche ni la compilation (c'est une notion *dynamique*) ni d'erreur de déroulement.

1.2 C'est fini [5 points]

Qu'affiche le code suivant ? **Justifiez** votre réponse.

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class A {
 5 public:
 6   virtual ~A() = default;
 7   virtual double f(int a) const = 0;
 8 };
 9
10 class B : public A {
11 public:
12   double f(int b) const {
13     if (b > 10) return 1.23;
14     return 98.76;
15   }
16 };
17
18 class C : public B {
19 public:
20   ~C() { cout << "Capri" << endl; }
21
22   double f(int c) const {
23     if (c > 100) return 5.65;
24     return -33.3;
25   }
26 };
27
28 int main()
29 {
30   C ta_c;
31   B* luga(&ta_c);
32   cout << luga->f( 25 ) << endl;
33   return 0;
34 }
```

Réponse et justification :
`-33.3`
`Capri`

`f()` est *virtuelle* (par héritage) et appelée au travers d'un *pointeur*, donc on a un comportement polymorphique.

Il y a appel direct de `C::~C()` pour détruire `ta_c` (variable usuelle, détruite directement).
Le pointeur `luga` en soit ne détuit rien.

Commentaire : `B::f()` et `C::f()` sont belles et bien virtuelles, même s'il n'y a ni `virtual`, ni `override` ; ces deux mots-clés sont optionnels.

Question 2 – Propriétaires et locataires [40 points]

2.1 Cadre général [20 points]

[...]

Début de réponse (n'oubliez pas de préciser dans quelle zone (A–F) vous ajoutez votre code) **:**

```
// ------------------------------------------------------------------------
// Zone A :
class Habitation;

// ------------------------------------------------------------------------
// Zone B : (Personne public)

void possede(Habitation* m) { if (m != nullptr) proprietes.push_back(m); }
void habite(Habitation& m); // ne peut pas encore être définie
                            // (à cause de la prédéclaration d'Habitation)

// ------------------------------------------------------------------------
// Zone C : (Personne private)

vector<Habitation*> logements;  // pas d'autre choix possible pour les pointeurs
vector<Habitation*> proprietes; // (utilisés comme références)
// Optionnel : un set<> me semble plus approprié ici

// ------------------------------------------------------------------------
// Zone D : (Habitation public)

Habitation(Personne& prop) : proprio(prop)
{ prop.possede(this); } // ne pas oublier de l'associer pour le propriétaire

void loge(Personne const& p) { habitants.push_back(&p); }

// ------------------------------------------------------------------------
// Zone E : (Habitation private)

const Personne& proprio;  // ou pointeur à la C (cf question 2.3)
vector<const Personne*> habitants; // utilisation comme référence, const optionnel

// ------------------------------------------------------------------------
// Zone F :

void Personne::habite(Habitation& m)
{
  logements.push_back(&m);
  m.loge(*this);
}
```

Alternatives :

`void possede(Habitation& m) { proprietes.push_back(&m); }`

On peut aussi accepter un `const` sur l'`Habitation` (mais il faut alors que la collection ait des `const` pointeurs).

On peut aussi choisir un pointeur pour `habite()` (mais pas de `const` ici !) et pour `loge()`.

Optionnel : une habitation pourrait aussi avoir l'ensemble de ses propriétaires. Il faut alors la maintenir de façon cohérente.

Par ailleurs, il ne faut surtout pas de destructeurs ! Tout d'abord, en dépit du nom, les propriétaires ne sont pas « propriétaires » au sens C++ du terme de leur habitations et les habitations ne sont pas « propriétaires » (au sens C++) de leur locataires ni de leur propriétaires (je ne souhaiterais pas habiter une telle habitation !).
Mais aussi et surtout, il n'y a *aucune* allocation dynamique dans le `main()` fourni (et faire ensuite des copies dynamiques n'a aucun sens (casse toute la cohérence des copies entre elles !)).

2.2 Lister les propriétaires et les locataires d'une population [15 points]

On souhaite maintenant fournir deux fonctions (`proprietaire()` et `locataires()` ; voir le code fourni précédemment) qui permettent, à partir d'une liste de personnes, d'extraire, l'une, la sous-liste des propriétaires présent dans la liste reçue en paramètre, et, l'autre, la sous-liste des locataires.

Ces sous-listes sont bien sûr des listes de *références* vers des personnes existant dans la population de départ, d'où le type de retour proposé dans le code fourni précédemment.

Donnez ici la définition de chacune de ces deux fonctions. Ajoutez si nécessaire des éléments aux classes définies précédemment (précisez quel code et dans quelle zone).

Réponses : L'idée principale ici est de « *déléguer* » les tests d'être propriétaire et d'être locataire à la classe `Personne` elle-même. Il ne faut surtout pas « sortir » (via un « *getter* ») les collections d'habitations pour les tester ensuite.

```
// ----------------------------------------------------------------------
// Zone B : (Personne public)

  bool est_proprio() const { return not proprietes.empty(); }

  bool est_locataire() const {
    // on est locataire dès qu'on habite chez qq1 d'autre
    // différence (ensembliste) non vide
    for (auto l : logements) {
      bool a_nous(false);
      for (auto p : proprietes)
        if (p == l) { // c'est chez nous
          a_nous = true;
          break; // optimisation
        }
      if (not a_nous) return true;
    }

    return false;
  }

// ----------------------------------------------------------------------
// Zone F :

vector<const Personne*> proprietaires(vector<Personne> const& liste)
{
  vector<const Personne*> retour;
  for (auto const& p : liste) if (p.est_proprio()) retour.push_back(&p);
  return retour;
}

vector<const Personne*> locataires(vector<Personne> const& liste)
{
  vector<const Personne*> retour;
  for (auto const& p : liste) if (p.est_locataire()) retour.push_back(&p);
  return retour;
}
```

On peut bien sûr utiliser des outils plus avancés (cf dernier cours) comme par exemple :

```
copy_if( liste.begin(), liste.end(), back_inserter(retour),
         [](Personne const& p) { return p.is_something(); }
       );
```

ou encore utiliser `find()` pour `est_locataire()`, ou même `set_difference()`.

Attention aussi à la définition de locataire : on n'a jamais dit qu'un propriétaire ne pouvait pas être locataire (cf `main()`) ou qu'une personne sans propriété était forcément locataire (elle pourrait être sans logement).

2.3 Achats et ventes [5 points]

Si l'on souhaitait pouvoir vendre des habitations, c.-à-d. les faire changer de propriétaire, **est-ce que cela changerait l'implémentation** que vous avez faite jusqu'ici ?

Si *oui*, **dites en quoi** (c.-à-d. où) et **pourquoi**.
Si c'est *non*, écrivez la méthode `vente()` des habitations, laquelle ferait changer leur propriétaire.

Réponse :

Oui, il faut changer la conception proposée car `Habitation::proprio` est une *référence* sur une `Personne` et ne peut donc pas être changée. Il faudrait mettre ici un *pointeur* ; un pointeur *à la C* ici puisque nous sommes dans le cas d'utilisation d'une référence (pas du tout d'allocation dynamique ici).
Par ailleurs, la personne pointée, elle-même, ne peut pas rester `const` car elle devra être modifiée (cf code ci-dessous).
Enfin il faut ajouter le code pour faire les modifications (cf ci-dessous).

Si un pointeur était déjà envisagé dès le départ, la réponse est alors « oui » ou « non » suivant que `Personne` est `const` ou `non` : le propriétaire ne peut plus rester `const` en tant que personne puisque son contenu (ses propriétés) doit être modifié.
Voici dans ce cas le code de `Habitation::vente()` :

```
void vente(Personne& p) {
    proprio->vend(this);
    p.possede(this);
    proprio = &p;
}
```

lequel nécessite donc aussi l'ajout d'une méthode à `Personne` pour pouvoir modifier ses propriétés (suppression)
(et donc, en toute rigueur, la réponse est finalement toujours « oui » `;-)`).

Commentaire : Beaucoup d'erreurs de différente nature sur cette Question 2 : mauvaise compréhention du cadre (on référence des objets existants), oubli de cohérence (des habitations qui ne sont pas référencées chez leur propriétaire, ou inversement), pas de pré-déclaration, beaucoup de « *getter* » inappropriés, ...

Question 3 – Algèbre linéaire [20 points]

Un élève envisage de faire un projet de programmation d'outils d'algèbre linéaire en C++. Il commence pour cela par mettre en place un cadre minimal factice, presque vide, pour tester l'organisation prévue pour ses fichiers. Il a donc produit les fichiers suivants (tels quels, avec les commentaires) :

`Makefile` :

```
1  CC = $(CXX)
2
3  all: algebre_lineaire
4
5  algebre_lineaire: algebre_lineaire.o matrice.o vecteur.o affichable.o
6
7  affichable.o: affichable.cc affichable.h
8  algebre_lineaire.o: algebre_lineaire.cc matrice.h affichable.h vecteur.h
9  matrice.o: matrice.cc matrice.h affichable.h vecteur.h
10 vecteur.o: vecteur.cc affichable.h vecteur.h matrice.h
```

④

`affichable.h` :

```
1  #pragma once
2  #include <iostream>
3
4  class Affichable {
5  public:
6    virtual void affiche(ostream&) const = 0;
7  };
8
9  ostream& operator<<(ostream& out, Affichable const& a);
```

① `std::`

`matrice.h` :

```
1  #pragma once
2
3  #include "affichable.h"
4  #include "vecteur.h"
5
6  class Matrice : public Affichable {
7  public:
8    // ...other things...
9
10   Vecteur operator*(Vecteur const& v);
11
12   virtual void affiche(std::ostream&) const override;
13
14 private:
15   // whatever...
16 };
17
18 class MatriceUnitaire : public Matrice {
19   // ...something...
20 };
```

② `class Vecteur;`

③ `const;`

vecteur.h :

```
 1 #pragma once
 2
 3 #include "affichable.h"
 4 #include "matrice.h"  ② class MatriceUnitaire;
 5
 6 class Vecteur : public Affichable {
 7 public:
 8   // ...other things...
 9
10   Vecteur rotation(MatriceUnitaire const& m) const;
11
12   virtual void affiche(std::ostream&) const override;
13
14 private:
15   // whatever...
16 };
17
```

affichable.cc :

```
 1 #include "affichable.h"
 2
 3 #include <iostream>
 4 using namespace std;
 5
 6 ostream& operator<<(ostream& out, Affichable const& a)
 7 {
 8   a.affiche(out);
 9   return out;
10 }
```

matrice.cc :

```
 1 #include "matrice.h"
 2 #include "vecteur.h"
 3
 4 #include <iostream>
 5 using namespace std;
 6                                    ③ const
 7 Vecteur Matrice::operator*(Vecteur const& v)
 8 {
 9   cout << "mat mult vect" << endl;
10   return Vecteur();
11 }
12
13 void Matrice::affiche(ostream& out) const
14 { out << "matrice"; }
```

vecteur.cc :

```
 1 #include "affichable.h"
 2 #include "vecteur.h"
 3 #include "matrice.h"
 4
 5 #include <iostream>
 6 using namespace std;
 7
 8 Vecteur Vecteur::rotation(
 9                 MatriceUnitaire const& m) const
10 {
11   cout << "rotation de " << *this
12        << " par " << m << endl;
13   return m * (*this);
14 }
15
16 void Vecteur::affiche(ostream& out) const
17 { out << "vecteur"; }
```

Ainsi que le fichier `algebre_lineaire.cc`, qui contient le `main()`. Il n'est pas donné ici car il n'est pas lui-même source d'erreur directe.

Une première tentative de compilation lui donne plein d'erreurs, mais dont voici les deux essentielles :

```
g++    -c -o algebre_lineaire.o algebre_lineaire.cc

In file included from matrice.h:3,
                 from algebre_lineaire.cc:1:
affichable.h:6:24: error: 'ostream' has not been declared
    6 |   virtual void affiche(ostream&) const = 0;
      |                        ^~~~~~~
In file included from matrice.h:4:
vecteur.h:10:20: error: 'MatriceUnitaire' has not been declared
   10 |   Vecteur rotation(MatriceUnitaire const& m) const;
      |                    ^~~~~~~~~~~~~~
```

À noter que la seconde erreur aurait aussi très bien pu être :

```
In file included from vecteur.h:4,
                 from algebre_lineaire.cc:2:
matrice.h:9:3: error: 'Vecteur' does not name a type
   10 |   Vecteur operator*(Vecteur const& v) const;
      |   ^~~~~~~
```

Que doit-il faire pour corriger ces **deux** erreurs ?

Apportez **directement vos corrections** sur le code donné précédemment, avec une brève explication à coté, et **indiquez ici** les lignes de quels fichiers vous avez corrigées :

Lignes corrigées : ① `affichable.h:6` et `9` (trois fois en tout)
② `vecteur.h:4` ; on pourrait aussi faire de même `matrice.h:4` ; mais l'un **ou** l'autre suffit à casser le cercle de dépendance ;

Après avoir suivi vos conseils et corrigé les deux erreurs ci-dessus, il en a une troisième :

```
g++    -c -o vecteur.o vecteur.cc
vecteur.cc: In member function 'Vecteur Vecteur::rotation(const MatriceUnitaire&) const':
vecteur.cc:11:20: error: passing 'const MatriceUnitaire' as 'this' argument
                         discards qualifiers [-fpermissive]
   13 |   return m * (*this);
      |                    ^
In file included from vecteur.cc:3:
matrice.h:10:11: note:   in call to 'Vecteur Matrice::operator*(const Vecteur&)'
   10 |   Vecteur operator*(Vecteur const& v);
      |           ^~~~~~~~
make[1]: *** [<builtin>: vecteur.o] Error 1
```

Que doit-il faire pour corriger cette erreur ?

Apportez **directement vos corrections** sur le code donné précédemment, avec une brève explication à coté, et **indiquez ici** les lignes de quels fichiers vous avez corrigées :

Lignes corrigées : ② `matrice.h:10` **et** `matrice.cc:7` : il faut que l'opérateur `*` soit `const`.

Commentaires : Pour ①, trop oublient de faire les **trois** corrections.

③ est trop souvent mal compris (et donc mal corrigé).

Il parvient donc finalement à compiler et aboutit à l'erreur suivante :

```
g++    -c -o algebre_lineaire.o algebre_lineaire.cc
g++    -c -o matrice.o matrice.cc
g++    -c -o vecteur.o vecteur.cc
g++   algebre_lineaire.o matrice.o vecteur.o   -o algebre_lineaire
/usr/bin/ld: vecteur.o: in function `Vecteur::rotation(MatriceUnitaire const&) const':
vecteur.cc:(.text+0x3b): undefined reference to `operator<<(std::ostream&, Affichable const&)'
/usr/bin/ld: vecteur.cc:(.text+0x62): undefined reference to
                                      `operator<<(std::ostream&, Affichable const&)'
collect2: error: ld returned 1 exit status
make[1]: *** [<builtin>: algebre_lineaire] Error 1
```

Que doit-il faire pour corriger cette erreur ?

Apportez **directement vos corrections** sur les fichiers donnés précédemment, avec une brève explication à coté, et **indiquez ici** les lignes de quels fichiers vous avez corrigées :

Lignes corrigées : C'est un problème d'édition de liens : il manque `affichable.o` dans la liste des fichiers objets ; il faut l'ajouter à la ligne 5 du `Makefile`.

Commentaire : Je suis surpris qu'en ayant fait le projet, beaucoup ne comprennent encore pas cette erreur (si fréquente pendant les séances).

Question 4 – Personnages et lieux [40 points]

On souhaite écrire une toute petite partie d'un programme de jeu ayant divers types de personnages (marchands, moines, ...) et divers types de lieux (magasins, bibliothèques, ...).
Les personnages ont la capacité de faire une action pour un lieu donné, et les lieux peuvent déclencher cette capacité pour un personnage donné. Par exemple, un marchand pour un magasin va faire une action de vente, un moine pour une bibliothèque l'action d'écrire, etc.

4.1 Marchands, moines, magasins et bibliothèques [25 points]

Définissez ci-dessous et sur la page suivante le code C++ permettant de mettre en œuvre la description précédente.
Concrètement, donnez tout le code nécessaire pour avoir des marchands et des moines (personnages), ainsi que des magasins et des bibliothèques (lieux). Pour simplifier, l'action d'un personnage dans un lieu sera simplement d'afficher un mot :

- un marchand pour un magasin affichera simplement `"vend"` ;
- un marchand pour une bibliothèque : `"lit"` ;
- un moine pour un magasin : `"achète"` ;
- un moine pour une bibliothèque : `"écrit"` ;

Remarque importante : même si c'est ici simplifié à l'extrême, il faut bien considérer ces actions comme de *vraies actions* qui devraient en réalité être bien plus compliquées et utiliseraient pleinement les caractéristiques de chaque personnage et de chaque lieu.

Réponse :

Voici une solution possible (page suivante).

On peut aussi le faire dans l'autre sens : « copier-coller » la définition de `Personnage::faire()`, laquelle appelle `Lieu::action()`, et définir les différentes actions dans les lieux.

Ce qui est important c'est de ne pas introduire de test de type de façon directe (`dyanmic_cast`, `typeid`) ou indirecte (attribut supplémentaire).

Commentaires : Certain(e)s ajoutent des collections de `Lieu` aux `Personnage` ou réciproquement. Ce n'est pas faux, mais pas nécessaire ici (peut être trop influencé(e)s par la Question 2 ?).

Trop d'oublis de : les pré-déclarations nécessaires, les destructeurs virtuels, la suppression des copies/affectations (pour `Jeu`).

Une autre erreur fréquente est d'oublier de passer l'argument à `faire()` ou à `action()`.

```
class Personnage;

class Lieu {
public:
  virtual void action(Personnage& quidam) = 0;
  virtual ~Lieu() = default;
};

class Magasin;
class Bibliotheque;

class Personnage {
public:
  virtual void faire(Magasin&)      = 0;
  virtual void faire(Bibliotheque&) = 0;
  virtual ~Personnage() = default;
};

// ==== La suite des lieux (maintenant que les Personnage sont complets)

class Magasin : public Lieu {
public:
  virtual void action(Personnage& quidam) override { quidam.faire(*this); }
};

class Bibliotheque : public Lieu {
public:
  virtual void action(Personnage& quidam) override { quidam.faire(*this); }
};

// ==== La suite des personnages (maintenant que les lieux sont connus)

class Marchand : public Personnage {
public:
  virtual void faire(Magasin& lieu) override
  { cout << "vend" << endl; }
  virtual void faire(Bibliotheque&  lieu) override
  { cout << "lit" << endl; }
};

class Moine : public Personnage {
public:
  virtual void faire(Magasin& lieu) override
  { cout << "Dans un magasin, le moine ACHETE." << endl; }
  virtual void faire(Bibliotheque&  lieu) override
  { cout << "Dans une bibliothèque, le moine ÉCRIT." << endl; }
};
```

4.2 Le jeu [15 points]

Définissez ci-dessous la classe `Jeu` (uniquement pour les aspects présentés dans le paragraphe d'introduction). On ajoutera simplement une méthode `tour()` qui permet de lancer (une fois) l'action de chaque personnage pour chaque lieu du jeu (cf exemple à la fin de la section suivante).

Réponse : Voici une solution possible, simple à écrire, mais « avancée » au niveau concepts du cours (« version 2b » du cours) :

```
class Jeu {
public:
  void tour() {
    for (auto& lieu : lieux)
      for (auto& perso : persos)
        lieu->action(*perso);
  }

  void ajoute(unique_ptr<Personnage> && p) { persos.push_back(move(p)); }
  void ajoute(unique_ptr<Lieu>       && l) { lieux.push_back(move(l)) ; }

private:
  vector<unique_ptr<Personnage>> persos;
  vector<unique_ptr<Lieu>>       lieux;
};
```

Note : l'appel pour `tour()` ne fonctionne pas dans l'autre sens :

```
perso->faire(*lieu);
```

car il n'y a pas de `Personnage::faire(Lieu&)` (à moins de tout inverser depuis le début et de rendre `Personnage::faire(Lieu&)` générique et spécialiser les `Lieux::action()`).

Voici une autre solution possible (toujours avec allocation dynamique, mais plus « risquée » car pas de contrôle que le pointeur reçu est bien alloué dynamiquement ; « version 2a » du cours) :

```
public:
  void ajoute(Personnage* p) { if (p != nullptr) persos.push_back(unique_ptr<Personnage>(p)); }
  void ajoute(Lieu*       l) { if (l != nullptr) lieux .push_back(unique_ptr<Lieu>      (l)); }

private:
  vector<unique_ptr<Personnage>> persos;
  vector<unique_ptr<Lieu>>       lieux;
```

Voici une troisième solution possible, avec copie polymorphique (« version 1 » du cours) :

```
public:
  void ajoute(Personnage const& p) { persos.push_back(p.clone()); }
  void ajoute(Lieu       const& l) { lieux .push_back(l.clone()); }

private:
  vector<unique_ptr<Personnage>> persos;
  vector<unique_ptr<Lieu>>       lieux;
```

Il faut alors bien sûr ajouter la copie polyphorphique aux `Personnage` et aux `Lieu` :

```
class Lieu {
public:
  virtual unique_ptr<Lieu> clone() const = 0;
};

class Personnage {
public:
  virtual unique_ptr<Personnage> clone() const = 0;
};

class Magasin : public Lieu {
public:
  virtual unique_ptr<Lieu> clone() const override
  { return unique_ptr<Lieu>(new Magasin(*this)); }
};

class Bibliotheque : public Lieu {
public:
  virtual unique_ptr<Lieu> clone() const override
  { return unique_ptr<Lieu>(new Bibliotheque(*this)); }
};

class Marchand : public Personnage {
public:
  virtual unique_ptr<Personnage> clone() const override
  { return unique_ptr<Personnage>(new Marchand(*this)); }
};

class Moine : public Personnage {
public:
  virtual unique_ptr<Personnage> clone() const override
  { return unique_ptr<Personnage>(new Moine(*this)); }
};
```

Enfin, une quatrième solution possible, avec « pointeurs à la C » (« version 3 » du cours) :

```
public:
  void ajoute(Personnage* p) { if (p != nullptr) persos.push_back(p); }
  void ajoute(Lieu*       l) { if (l != nullptr) lieux .push_back(l); }

  ~Jeu() {
    for (auto p : persos) delete p;
    for (auto l : lieux ) delete l;
  }
  Jeu& operator=(const Jeu&) = delete;
  Jeu(const Jeu&)            = delete;
  Jeu()                      = default; // remettre le cteur par défaut

private:
  vector<Personnage*> persos;
  vector<Lieu*>       lieux;
```

Il faut alors bien sûr ne pas oublier de libérer la mémoire (dans le destructeur), et gérer la copie et l'affectation (p.ex. les supprimier)

Enfin, on peut bien sûr utiliser d'autres containers plus appropriés tels que les `set` (mais il faudrait alors avoir un `operator<`, que l'on a pour les pointeurs ; ou sinon `unordered_set`).

Complétez enfin le `main()` suivant pour ajouter au `Jeu j` un marchand, un moine, un magasin et une bibliothèque, chacun alloué dynamiquement :

```
int main()
{
  Jeu j;
  // Ajoutez ici **DYNAMIQUEMENT** un marchand, un moine, un magasin et une bibliothèque :

  j.ajoute(make_unique<Moine>());
  j.ajoute(make_unique<Marchand>());

  j.ajoute(make_unique<Magasin>());
  j.ajoute(make_unique<Bibliotheque>());

  j.tour(); // lancement d'un tour de jeu

  return 0;
}
```

Ce programme devrait donc afficher (peut être dans un ordre différent) :

```
vend    lit    achète    écrit
```

Ci-dessus la version pour la première solution proposée précédemment.
Il faut bien sûr adapter ces appels à la solution choisie plus haut :

- versions 2 et 4 :

```
j.ajoute(new Moine());
j.ajoute(new Marchand());
j.ajoute(new Magasin());
j.ajoute(new Bibliotheque());
```

- version 3 :

```
j.ajoute(Moine());
j.ajoute(Marchand());
j.ajoute(Magasin());
j.ajoute(Bibliotheque());
```