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, sections MA et PH
Chappelier J.-C.

Programmation Orientée Objet (SMA/SPH) : Correction examen final

6 juillet 2021

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 de trois heures pour faire cet examen (16h15 – 19h15).
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**.
   Vous disposez, si nécessaire, d'une page blanche supplémentaire en fin de sujet.
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 des assistants.
6. L'examen comporte cinq exercices indépendants (sur **??** pages), qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose :
   — questions « courtes » : 29 points ;
   — aide à la programmation : 35 points ;
   — conception : 28 points ;
   — trouver les erreurs : 24 points ; et
   — déroulement de programmes : 14 points.

   Le total est de 130 points ; tous les exercices comptent pour la note finale.

Question 1 – Plus ou moins courtes questions de cours [29 points]

1.1 Échauffement [1.5 points]

Que se passe-t-il si on écrit le code suivant dans la déclaration d'une classe :

```
virtual double f(int) const = 0;
```

a) Le code n'est pas correct.

b) Il n'est pas possible de redéfinir la méthode `f()` dans les sous-classes.

c) On s'attend à ce que cette classe ait au moins une sous-classe.

d) Il n'est pas possible de créer des objets de cette classe.

Entourez la ou les réponse(s) correcte(s). Pénalité en cas de sélection d'une mauvaise réponse.

Réponses **c)** et **d)**.

Réussie à 71.8%.

1.2 Pas très solide [3 points]

Le programmeur d'une classe `Solide` a défini l'attribut `masse` comme ci-contre. Un utilisateur de cette classe peut donc accidentellement affecter une masse négative ; ce qui n'a pas de sens.
Expliquez comment améliorer cette classe pour éviter ce problème. Quelle(s) modification(s) suggérez-vous (explication sous forme de texte, pas de code) ?

```
class Solide {
  // ...(autres choses)...
public:
  double masse;
};
```

Réponse :

- l'attribut doit être `private` (ou `protected`)
- constructeur (en tout cas) et « méthode set » si vraiment nécessaire (et sinon, pourquoi ne pas mettre l'attribut en `const` ?)
- « méthode get » si nécessaire (dépend de comment la masse sera utilisée ; pourrait être utilisée strictement en interne (dans d'autres méthodes))

Commentaires de correction : réussie à 70.9%. Les principales erreurs ont été :

- diverses solutions *sans* mettre la masse en `private`
- des `unsigned double` (???)

1.3 B ? A ? Bah... [9 points]

Considérant les déclarations suivantes :

```
class A {
public:
          int f1() const { return 1    ; }
  virtual int f2() const { return 2    ; }
  virtual int f3() const { return f1() ; }
          int f4() const { return f2() ; }
};

class B : public A {
public:
  int f1() const          { return 101  ; }
  int f2() const override { return 202  ; }
  int f3() const override { return f2() ; } //!\ f2
  int f4() const          { return f1() ; }
};

B babe;
A aha;
A* ptr1(&babe);
B* ptr2(&babe);
A* ptr3(&aha);
```

quelle est la valeur des expressions suivantes ? Indiquez « *erreur* » si l'expression ne compile pas ou est erronée.
Dans tous les cas, justifiez brièvement votre réponse.

a) `ptr1->f1()` : 1 ; `ptr1` pointe sur une instance de `B`, *mais* `ptr1` est un pointeur de type `A*` et `f1()` n'est pas virtuelle.

b) `ptr3->f1()` : 1 ; `ptr3` pointe sur une instance de `A`

c) `ptr1->f2()` : 202 ; `ptr1` pointe sur une instance de B et `f2()` est virtuelle

d) `ptr1->f3()` : 202 ; `ptr1` pointe sur une instance de `B`, et `f3()` est virtuelle (Cela suffit. On a été gentil ici et fait un appel à `f2()` qui est aussi virtuelle, mais même si on avait fait appel à `f1()`, c'est bien `B::f1()` qui eût été appelée ici).

e) `ptr2->f3()` : 202 ; `ptr2` pointe sur une instance de `B`

f) `ptr3->f3()` : 1 ; `ptr3` pointe sur une instance de `A`

Réussie à 89.0%.

1.4 Constructions [4 points]

Considérant les *extraits* de code à droite :

```
class A {
public:
  A() = default;
  A(double);
};

double f1(A);
double f2(A const&);
// ...
A un_a;
double x;
// ...
x = f1(un_a);
// ...
x = f2(un_a);
```

① Quel constructeur est utilisé lors de l'appel `f1(un_a)` ?

a) `A::A()`

b) `A::A(double)`

c) `A::A(A const&)`

d) Aucun.

e) La question n'a pas de sens, un tel appel ne peut pas compiler.

Justifiez *brièvement* votre réponse : **c)**. On appelle le constructeur de copie pour passer l'argument par valeur à la fonction.

② Quel constructeur est utilisé lors de l'appel `f2(un_a)` ?

a) `A::A()`

b) `A::A(double)`

c) `A::A(A const&)`

d) Aucun.

e) La question n'a pas de sens, un tel appel ne peut pas compiler.

Justifiez *brièvement* votre réponse : **d)**. L'argument de `f2(x)` est passé par référence, donc il n'y a *aucune* copie de faite.

Réussie à 63.9%.

1.5 Vécu [6.5 points]

① **[2.5 points]** Le programme en page ci-contre compile-t-il et s'exécute-t-il correctement ? Si oui, dire ce qu'il affiche, sinon justifiez pourquoi.
Réponse : Oui ; il affiche « `12 autres choses` » (polymorphisme sur `affiche()`).

② **[4 points]** Si l'on suppose que l'on peut l'exécuter, combien de `double` ce programme crée/utilise-t-il en tout ? Justifiez votre réponse. (Vous pouvez continuer à répondre sur la page suivante.)
Réponse : 48 : 4 fois 12 : 12 pour `a`, 12 pour la copie de `a` passé au constructeur de `c`, et `c` lui-même en contient 2 (fois 12) : un direct et un par héritage.

Commentaires de correction : réussie à 42.3%. Les principales erreurs ont été :
- manque de justifications de l'affichage (vu l'ambiguité de la formulation (« sinon »), j'en ai réduit d'autant le barème)
- beaucoup pensent que le code ne compile pas, pour diverses fausses raisons, la plus fréquente étant de croire que le constructeur de copie par défaut n'est plus présent à cause de la présence du constructeur écrit (ce n'est *que* le constructeur *par défaut* par défaut (2 fois) qui est supprimé) ;
- concernant le compte, beaucoup ratent la copie et surtout la partie héritée
- d'autres comptes trop de `double`, certain(e)s par mauvaises connaissances de l'exécution du programme (confusion entre variable et expression) et d'autres par l'ambigüité de la formulation (« utilise ») ; nous n'avons pas compté comme faux ces imaginaires `double` supplémentaires.

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

class ChampPotentiels {
public:
  ChampPotentiels(size_t taille)
    : contenu(taille)
  {
    for(size_t i(0); i < taille; ++i)
    { contenu[i] = 7.0 * sin(12.0*i + 0.75); }
  }

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

  virtual void affiche() const
  { cout << size() << " potentiels" << endl; }

private:
  vector<double> contenu;
};

class Ciel : public ChampPotentiels {
public:
  Ciel(ChampPotentiels x)
   : ChampPotentiels(x), autre_chose(x.size())
  {
    for(size_t i(0); i < autre_chose.size(); ++i)
    { autre_chose[i] = 2.236 * sqrt(pow(i, 2.0) + pow(2*i, 2.0)); }
  }

  virtual void affiche() const override
  { cout << autre_chose.size() << " autres choses" << endl; }

private:
  vector<double> autre_chose;
};

int main()
{
  ChampPotentiels a(12);
  Ciel c(a);
  c.affiche();
  return 0;
}
```

suite au dos ☞

1.6 Qu'en dites vous ? [5 points]

① **[2.5 points]** Le programme ci-contre compile-t-il et s'exécute-t-il correctement ?
Si oui, dire ce qu'il affiche.
Dans tous les cas, justifiez votre réponse.
Réponse :

② **[2.5 points]** Peut-on faire l'appel : `v.g(1.1)` ?
Si oui, dire ce qu'il affiche.
Dans tous les cas, justifiez votre réponse.
Réponse :

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

class A {
public:
  A(double x) : x_(x) {}

  void f(double a) const
  { cout << a + x_ << endl; }

  void g(double a) const
  { cout << x_ * a << endl; }

  virtual void g(const string& s) const
  { cout << s << 50.0 * x_ << endl; }

protected:
  double x_;
};

class B : public A {
public:
  using A::A;

  void g(const string& s) const override
  { cout << s << 2.0 * x_ << endl; }
};

int main() {
  A u(3.0);
  B v(7.0);
  u.g("val1 = ");
  v.g("val2 = ");
  v.f(1.1);
  return 0;
}
```

① Ce code compile (héritage de `f` (et `g`)) et s'exécute correctement. Il affiche :

```
val1 = 150
val2 = 14
8.1
```

Pas de polymorphisme ici car pas de pointeurs ni de référence, donc appel direct de la méthode de la classe elle-même : donc $50 \times 3 = 150$ pour `u`, , et $2 \times 7 = 14$ pour `v` (et trivialement $7 + 1.1$ pour `v.f()`).

② Non car « il n'y a surcharge que dans la même portée » : dans la portée de `B`, `B::g(const string&)` **masque** `A::g(double)`.

Commentaires de correction : réussie à 30.4%. Les principales erreurs ont été :
- manque de justification (d'où viennent les valeurs indiquées, pourquoi `v` a bien un `f`)
- et beaucoup n'ont pas du tout compris ce qui se passe vraiment en ②.

Question 2 – Et si vous deveniez assistant(e) ? [35 points]

2.1 You know what ? [8 points]

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

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

class Home {
public:
  Home() : companion(nullptr) {}
  void adopt(Pet* p) { companion = p; }
private:
  Pet* companion;
  Home& operator=(Home const&) = delete;
  Home(Home const&)            = delete;
};

class Dog : public Pet {
public:
  void eat()
  { cout << "Crunch!"    << endl; }
  void talk()
  { cout << "Woof woof!" << endl; }
};

int main()
{
  Home sweet_home;    Dog d("Droopy");
  // Quoi faire ?...


  return 0;
}
```

Dans un exercice qui fournissait les classes `Pet` et `Home` (inchangées), un étudiant est parvenu au code ci-contre, mais il se trouve bloqué dans l'écriture du `main()` car il ne sait pas comment écrire l'adoption du `Dog d` par `sweet_home`. Il vous demande de l'aide en faisant la réflexion suivante :
« Je voudrais mettre `d` dans `sweet_home`, mais il n'y a de la place que pour un `Pet`... Ça ne joue pas ! »

① Qu'avez-vous à lui répondre ?

② Écrivez (dans le code C++ ci-contre) une solution pour que `sweet_home` adopte `d`.

③ Comment appelle-t-on une classe comme la classe `Pet` ?
Pourquoi l'appelle-t-on ainsi ?

④ Le code écrit ci-contre par l'étudiant (hors classes `Pet` et `Home`) ne compile pas. Sauriez-vous dire pourquoi (une seule raison) ?

Réponses :

① Pas de problème : un `Dog` **est un** `Pet` (héritage), aussi au sens des types de C++.

② `sweet_home.adopt(&d);`

③ Comme elle a au moins une méthode virtuelle pure, c'est une classe *abstraite*. On l'appelle ainsi parce qu'on ne peut pas en créer d'instance (elle ne peut pas se «*concrétiser*» (en tant que telle)).

Note : on pourrait aussi imaginer répondre : une *super-classe* parce que `Dog` en hérite (sous-classe) ; mais c'est quand même moins spécifique.

④ Parce que `Dog` n'a pas de constructeur prennant de `string` (on n'hérite pas les constructeurs ; elle ne possède donc que le constructeur de copie par défaut (et celui de déplacement). A noter que le constructeur par défaut par défaut n'existe pas non plus pour `Dog` puisque `Pet` n'a pas de constructeur par défaut).

Commentaires de correction : réussie à 65.6%. Les principales erreurs ont été :
— ne pas comprendre la question ① (ceci dit, nombreux sont celles et ceux qui ensuite répondent correctement à ②)

— penser qu'il faille passer par une variable suplémentaire de type `Pet*` pour ②; ceux/celles là n'ont pas bien compris le sens (sémantique; pas direction) de « est-un »
— et beaucoup trop font des `new` dans ②

④ est bien réussie.

En raison du fort taux de la première erreur citées ci-dessus (lié à la formulation trop imprécise de la question ①), j'ai décidé de réduire mon barème d'autant (= transformer en bonus). Le taux de réussite est alors de 77%.

2.2 Ça ne compile pas ! [14 points]

Un autre étudiant vous interpelle. Il a écrit le code suivant :

```
1  class Vector2D {
2  public:
3    Vector2D(double a = 0.0, double b = 0.0) : x(a), y(b) {}
4
5    bool operator==(const Vector2D& autre) const
6    { return ((x == autre.x) and (y == autre.y)); }
7
8    double norme2() const
9    { return double(x*x + y*y); }
10
11   double norme() const
12   { return double(sqrt(x*x + y*y)); }
13
14   // vecteur unitaire colineaire et de meme sens
15   Vector2D& operator~() const
16   { return Vector2D(x / norme(), y / norme()); }
17
18 private:
19   double x, y;
20 };
```

qui provoque cette erreur de compilation :

```
vecteur_classe.cc: In member function 'Vector2D& Vector2D::operator~() const':
vecteur_classe.cc: error: cannot bind non-const lvalue reference of type
                          'Vector2D&' to an rvalue of type 'Vector2D'
   16 |    { return Vector2D(x / norme(), y / norme()); }
      |             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

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

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

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

— Ne pas oublier `operator!=()`.
— Ne pas comparer des `double` avec ==.
— Pas de copié-collé (`norme()` et `norme2()`).
— Protéger `operator~()` contre le vecteur nul (division par 0).
— (mineur, stlye) il est inutile de mettre les `double` lignes 9 et 12.
— (mineur) Mettre `operator~()` en externe puisque n'utilise que du `public`.

(place pour encore plus de conseils si nécessaire.)

Plus loin dans son code vous voyez la chose suivante :

```
1  vector<double> vitesse = calcule_vitesse(machin);
2
3  if (sqrt(vitesse[0]*vitesse[0]+vitesse[1]*vitesse[1]+vitesse[2]*vitesse[2])
4      >= vitesse_vent) {
5    faire_quelque_chose(vitesse, machin);
6  } else {
7    faire_autre_chose(vitesse, machin);
8  }
```

③ **[7 points]** Quel(s) autres conseil(s) voudriez-vous lui donner ?
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).

— Pourquoi utiliser `vector` plutôt que `array` alors que la taille est connue *a priori* et ne change pas ?
— Encore mieux : pourquoi ne pas en faire une classe ? qui utiliserait/dont hériterait `Vector2D`...
— Écrire au moins une fonction et sinon une méthode pour la norme.
— Au final, on pourrait même carrément écrire simplement une méthode pour `machin`, vu que tout ne dépend que de lui (et `machin` utiliserait en interne des `Vector3D`).

Tout ce code deviendrait alors simplement :

```
machin.faire_en_fonction_du_vent(vitesse_vent);
```

Commentaires de correction : réussie à 19.4%. Les principales erreurs ont été :
— ② pratiquement personne ne pense à `operator!=()`
— ② très peu pensent à ne pas comparer des `double` avec ==
— ② plein de critiques peu ou pas du tout fondées
— ③ 1 seule personne (!) pense assez loin au point d'en faire une méthode de `machin`
— ③ mauvaise utilisation de `norme()` (comme une méthode de `vector<double>`)
— ③ diverses critiques peu ou pas fondées

Même si je ne m'attends en effet pas à ce que la majeure partie de la classe devienne assisant(e), j'ai été surpris par le très faible taux de réussite de cette sous-question (certes difficile !) et de la suivante. J'ai donc revu mon barème à la baisse en conséquence.

suite au dos ☞

2.3 C'est bien ? [13 points]

Une partie d'un projet de groupe (fictif !!) peut se résumer par le code suivant :

```
1  #include <iostream>
2  using namespace std;
3
4  class Dessinable {
5  public:
6    virtual void dessine() const = 0;
7  };
8
9  class Montagne : public Dessinable {
10 public:
11   void dessine() const {
12     cout << "Une montagne" << endl;
13   }
14 };
15
16 class System : public Dessinable {
17 public:
18   System(Montagne* pm) : m(pm) {}
19   ~System() { delete m; }
20   void dessine() const {
21     cout << "Un systeme" << endl;
22   }
23   Montagne* get_montagne() const { return m; }
24
25 private:
26   Montagne* m;
27 };
28
29 int main() {
30   Montagne m;
31   System s(&m);
32   s.dessine();
33   s.get_montagne()->dessine();
34   return 0;
35 }
```

① Ce code compile-t-il et s'exécute-t-il correctement ? Si oui, qu'affiche-t-il ?
Justifiez votre réponse.

Il compile, mais ne s'exécute pas totalement correctement en raison du `delete` (ligne 19) de la `Montagne` allouée statiquement (ligne 30). Il est fort probable qu'il ait le temps d'afficher correctement le système et la montagne avant de planter.

② Quel(s) conseil(s) voudriez-vous donner à ce groupe ?
Réécrivez (ensuite, après les conseils) le code C++ que vous leur proposez (uniquement les parties proposées, ne récrivez pas le code inchangé).

Il y a deux *grosses* erreurs de conception et une « standard » :

1. la non-clarification de la propriété de la montagne : pourquoi `delete` si ce n'est pas le `System` qui est propriétaire ?

   Une solution *possible* (il y en a d'autres) serait d'avoir une *référence* (`const` ?) vers la `Montagne`, au lieu d'un pointeur.

   (Changer les lignes 18, (23,) 26 et 31 (et 33) en conséquence et supprimer ligne 19.)

   Ou alors décider que c'est le `System` qui est propriétaire et

— soit construire en interne une « vraie » `Montagne` (en passant les paramètres nécessaires au constructeur de `System`) ;
— soit partir sur de la copie dynamique, voire polymorphique si nécessaire (si polymorphisme des `Montagnes`).

En cas de non-propriété de la `Montagne` par le `System`, il faut simplement supprimer le destructeur (ne pas en mettre du tout ; celui fourni par défaut suffit).

2. La fuite d'encapsulation de la `Montagne` du `System` ;
   on pourrait simplement *déléguer* le dessin des montagnes au `System` :

```
void System::dessine() const {
    cout << "Un systeme" << endl;
    m->dessine();
}
```

   la ligne 33 du `main()` étant alors simplement supprimée (ainsi que, bien sûr, le `get()` de la `Montagne` du `System`).

3. La classe `Dessinable` a pour essence même d'être polymorphique ; elle *doit* donc donner la possibilité de faire une destruction polymorphique : il *faut* mettre son destructeur comme virtuel ; p.ex. :
   `virtual ~Dessinable() = default;`

De façon plus mineure, mais liée au point 1 : en raison de la présence du pointeur, ou simplement d'avoir fait un destructeur (« règle des trois »), il faudrait se poser la question de la copie du `System`.

Le mieux, me semble-t-il, est de l'interdire :

```
System(const System&) = delete;
System& operator=(const System&) = delete;
```

Et encore plus mineur (style), on pourrait marquer comme `override` (voire rajouter le `virtual`) au redéfinitions de `dessine()`.

Commentaires de correction : réussie à 9.7%. Les principales erreurs ont été :
— je suis assez déçu que presque personne ne voit le segmentation fault ici. C'est que j'ai raté quelque chose dans mon enseignement...
— peu de choses pertinentes sur ② mis à part le destructeur virtuel vu assez souvent et les deux conseils mineurs cités dans le corrigé.

Comme dit dans la sous-question précédente, j'ai donc revu mon barème à la baisse en conséquence du très faible taux de réussite sur cette sous-question atypique.

Question 3 – Conception [28 points]

On souhaite modéliser informatiquement la vente de billets à des files de clients de spectacles. Pour simplifier, on s'intéresse *uniquement* aux aspects décrits ci-dessous et l'on ignore tout le reste (on ne se préoccupe en particulier ni de l'argent possédé par les personnes, ni du prix des places, ni du nombre de places disponibles).

Dans la file d'attente, il y a des personnes. Chaque personne a le titre du spectacle qu'elle voudrait voir, et le nombre de places qu'elle veut acheter.

En plus des personnes « standard », il y a deux sortes de personnes spécifiques :
- des salariés, qui ont en plus le nom de leur entreprise ;
- des retraités, qui ont en plus leur âge.

Pour simplifier, il n'y a pas de personne retraitée qui soit salariée.

Chaque personne peut « se présenter », d'une façon qui dépend de sa nature (standard, salarié ou retraité) :
- les personnes standard disent simplement :
  « *Bonjour, je voudrais* `[nombre]` *billet(s) pour* `[titre]`. »
- les employés disent :
  « *Bonjour, je voudrais* `[nombre]` *billet(s) pour* `[titre]`.
  Je travaille chez `[entreprise]` ; *faites-vous des réductions ?* »
- et les retraités disent :
  « *Bonjour, je voudrais* `[nombre]` *billet(s) pour* `[titre]`.
  J'ai `[age]` *ans ; ai-je droit à un tarif spécial ?* »

Du coté de la vente/salle de spectacle, il faut pouvoir :
- ajouter une personne à la (fin de la) file d'attente ;
- vendre les billets qu'elle veut à une personne ;
- gérer la file en s'occupant de la première personne dans la file (détails en sous-question ②).

① **[18.5 points] Conception**

Sans donner tout le détail du code complet (on ne demande ici qu'une *conception*), écrivez **en C++** les classes, les éventuelles relations d'héritage, les attributs et les méthodes des classes, les droits d'accès et les éventuelles fonctions (externes) que vous utiliseriez pour implémenter un tel programme.

Précisez les types des attributs et les prototypes des méthodes/fonctions, mais ne donnez pas leur définition (on répète : il s'agit ici de la partie *conception*, pas de l'implémentation ; c.-à-d. les prototypes, pas les définitions des méthodes).

Lorsque c'est nécessaire, indiquez également les constructeurs et les destructeurs (sans leur définition).

② **[9.5 points] Programmation**

a) Implémentez (c.-à-d. définissez) la méthode ou la fonction qui s'occupe de la première personne dans la file. Cette méthode ou fonction doit (s'il y a quelqu'un) :

a) la faire « se présenter » (comme expliqué plus haut) ;

b) lui vendre les billets qu'elle désire ;

c) la retirer de la file.

b) Écrivez ensuite la ou les fonction(s) ou méthode(s) qui permettent aux différentes personnes de « se présenter ». Vous pouvez si nécessaire abréger le *texte* effectivement écrit, mais sans négliger/retirer les paramètres/attributs nécessaires.

Réponses :

(suite des réponses à la Question 3)

Voici *une* réponse possible :

```
class Personne {
public:
  Personne(const string& titre_, // string_view est encore mieux
           unsigned int nb_places = 1); // option optionnelle ;-)
  virtual ~Personne() = default;

  virtual void presenter() const;

private:
  string titre; // pourrait être une const-référence : on partage le titre
  unsigned int nb;
};

class Salarie : public Personne {
public:
  Salarie(const string& patron,
          const string& titre_, unsigned int nb_places = 1);
  virtual ~Salarie() = default; // optionnel (pas de sous-classe connue)

  virtual void presenter() const override; // virtual et override optionnels

private:
  string employeur; // pourrait être const (limite) ;
                    // pourrait être une const référence (limite aussi)
};

class Retraite : public Personne {
public:
  Retraite(int age_, string titre_, unsigned int nb_places = 1);
  virtual ~Retraite() = default; // optionnel (pas de sous-classe connue)

  virtual void presenter() const override; // virtual et override optionnels

private:
  int age;
};

class FileSpectacle {
public:
  void au_suivant();
  void arrive(const Personne * p);

  void vendre_billets(const Personne *) const; // const optionnel

private:
  deque<const Personne *> file; // vector toléré au niveau de ce cours
                                // on peut aussi utiliser queue<>
  // fonction outil optionnelle (typiquement si pas deque)
  void supprimer_premier() { file.pop_front(); }
};
```

J'ai mis tous les attributs en `private`, mais on peut les tolérer en `protected`.

Pour toutes les classes : s'il y a des attributs, il faut le constructeur pour les initialiser ; s'il n'y a pas d'attribut (ou si l'attribut a un constructeur par défaut qui suffit), le constructeur par défaut par défaut suffit (sauf pour les sous-classes dont la super-classe a un constructeur spécifique, mais il n'y en a pas ici).

Pour la classe `FileSpectacle` : vu le polymorphisme des `Personne`, il faut des pointeurs ; mais je ne vois pas ici d'autre solution que des « pointeur à la C » (ou alors des `shared_ptr`, mais pas utilisés en cours) car nous somme clairement dans « le cas numéro 1 » du cours : référence vers les personnes existant par ailleurs.

Commentaires de correction : réussie à 96% (bravo ! !). Les principales erreurs ont été :
- oubli du destructeur virtuel de `Personne` (ce qui me surprends après 2.3.②)
- les `const` des méthodes qui doivent l'être

② a)

```
void FileSpectacle::au_suivant() {
  if (not file.empty()) {
    file[0]->presenter();
    vendre_billets(file[0]);
    supprimer_premier(); // simple pop_front() si deque

  } else { // PAS DEMANDÉ
    cout << "Y'a p'us personne !" << endl;
  }
}
```

② b)

```
void Personne::presenter() const {
  cout << "Bonjour, je voudrais "
       << nb << " billet(s) pour " << titre << endl;
}

void Salarie::presenter() const {
  Personne::presenter();
  cout << "Je travaille chez " << employeur
       << " ; vous faites des réductions ?" << endl;
}

void Retraite::presenter() const {
  Personne::presenter();
  cout << "J'ai " << age << " ans ; "
       << "ai-je droit à un tarif spécial ?" << endl;
}
```

Commentaires de correction : réussie à 70%. Les principales erreurs ont été :
- la difficulté à gérer la première/dernière personne (notion de file) ; peu pensent (lorsque nécessaire) à déléguer à une méthode auxiliaire
- certain(e)s inventent des méthodes à `vector`
- ne pas gérer la file vide
- plusieurs personnes ont verifié (dans `suivant()`) si les pointeurs étaient `nullptr`, alors que celà devrait être garanti par conception de `ajouter()` ; il semble qu'il y ait ici une mauvaise compréhention des collections de pointeurs (et de la notion de « propriété »
- (inutile, mais surtout) mauvaise utilisation de `delete` comme dans `file[0] = delete`
- peu d'utilisation de `queue` (dernier cours) ou de `deque` (non présenté, donc ça c'est normal)
- des incohérence entre ce qui est proposé en ① et ce qui est codé en ②

Par contre, je suis content que la non-duplication de code dans `presenter()` ait correctement été faite dans la majorité des copies.

Question 4 – Trouver les erreurs [6x4=24 points]

Les six *extraits* de programmes suivants contiennent chacun une erreur. Pour chaque programme, indiquez (sur le programme ou à coté) où se situe l'erreur, expliquez pourquoi c'en est une et comment la corriger.

Attention ! On ôtera 1 point pour toute indication d'une erreur qui n'en est pas une.

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5  public:
6    A(double x = 1.0) : x_(x) {}
7    double operator*(const A& autre) const {
8      return x_ * autre.x_;
9     }
10 private:
11   double x_;
12 };
13
14 int main() {
15   A a1(4.0);
16   A a2();
17   cout << a1 * a2 << endl;
18   return 0;
19 }
```

Réponse : Ligne 16 : `a2` n'est pas une instance de `A`, mais une fonction (qui ne prend pas d'argument et qui retourne un `A`).
Correction : supprimer les « `()` » : `A a2;`
Réussie à 28.6%.

```
1  class A {
2  public:
3    A(double x = 0.0) : x_(x) {}
4  protected:
5    double x_;
6  };
7
8  class B : public A {
9  public:
10   B(double x) : A(x) {}
11   void f(A const& autre) {
12     cout << "partie A ";
13     if (abs(autre.x_ -x_) < 1e-10)
14       cout << "identique";
15     else
16       cout << "differente";
17     cout << endl;
18    }
19 };
```

Réponse : Ligne 13 : on ne peut accéder aux attributs `protected` *que* dans le rôle « d'héritier », pas comme accès externe à la classe (p.ex. paramètre `autre` ici).
Correction : ajouter à `A` un « getter » pour `x_`, ou une méthode
`double distance(double y) const`,
voire `double distance(A const& a) const`,
qui retourne `abs(x_ - y)`
(ou `abs(x_ - a.x_)` ; là on est dans la bonne portée !).
Réussie à 58.2%.

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5  public:
6    A(double x = 0.0) : x_(x) {}
7    virtual double f(double y) const = 0;
8    virtual double g(double y) const;
9    virtual ~A() = default;
10 protected:
11   double x_;
12 };
13
14 class B : public A {
15 public:
16   B(double x) : A(x) {}
17   double f(double y) const { return g(x_ + y); }
18 };
19
20 int main() {
21   B b(1.0);
22   cout << b.f(2.0) << endl;
23   return 0;
24 }
```

Réponse : Ligne 8 : la méthode `g()` n'est pas définie.
Corrections possibles : soit lui donner un corps ; soit la mettre comme virtuelle pure, mais il faut alors la définir dans `B` ; soit, finalement, totalement supprimer cette ligne vu qu'il n'y a aucun `g()` nulle part ailleurs.
Réussie à 44.1%.

```
1  class A {
2  public:
3    A(double x = 0.0) : x_(x) {}
4    virtual void f() const = 0;
5    virtual void g() { x_ *= 2.0; }
6    virtual ~A() = default;
7  protected:
8    double x_;
9  };
10
11 class B : public A {
12 public:
13   void g() { x_ *= 3.0; }
14 };
15
16 int main() {
17   B u;
18   u.g();
19   return 0;
20 }
```

Réponse : Variante de l'erreur précédente : ligne 14 : la méthode `f()` n'est pas définie, donc `B` reste une classe abstraite dont on ne peut pas créer d'instance.
Correction : définir `f()` dans `B`.
(Note : supprimer les lignes 17 et 18 serait aussi une correction possible, mais absurde ! (la correction correspondante devrait alors aussi supprimer les lignes 1 à 15 `;-)`))
Réussie à 74.0%.

```
1  class A {
2  public:
3    A(double x) : x_(x) {}
4  private:
5    double x_;
6  };
7
8  class B : public A {
9  public:
10   B(double y) : y_(y) {}
11 private:
12   double y_;
13 };
```

Réponse : Ligne 3 **ou** ligne 10 : `A` n'a pas de constructeur par défaut.
Corrections possibles : ajouter un constructeur par défaut à `A` (p.ex. en ajoutant une valeur par défaut au paramètre de son constructeur)
ou
faire un appel explicite (correct) au constructeur de `A` dans la « section deux-points » du constructeur de `B`.
Réussie à 80.2%.

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5  public:
6    A(double x): x_(x) {}
7    virtual double f(double y);
8    virtual double g(A const& autre,
9                     double y) const;
10   virtual ~A() = default;
11 protected:
12   double x_;
13 };
14
15 double A::f(double y)
16 { return x_ + y; }
17
18 double A::g(A const& autre, double y) const
19 { return f(autre.f(y)); }
20
21 int main() {
22   A a(1.0);
23   A b(2.0);
24
25   cout << b.g(a, 3.0) << endl;
26
27   return 0;
28 }
```

Réponse : Ligne 7 (et 15) : la méthode `f()` de `A` n'est pas `const`.
Remarque : c'est *doublement* une erreur car `autre` est une référence constante, mais aussi car `g()`, qui est `const`, appelle `f()`.
Correction : ajouter `const` lignes 7 et 15.
Réussie à 47.1%.

Indication : pour ce dernier programme, le compilateur indique ceci :

```
prog.cc: In member function 'virtual double A::g(const A&, double) const':
prog.cc:19: error: passing 'const A' as 'this' argument discards qualifiers
   19 |   return f(autre.f(y));
      |                    ^
```

Commentaires de correction : globalement réussie à 42.3%. Les principales erreurs ont été :

- mal lire la consigne et ne pas donner la correction
- ne pas voir l'erreur
- trop d'approximations dans le sixième code (où pourtant le message d'erreur est donné)

Mais la relation d'héritage et ses conséquences ont été bien assimilées par la plupart des étudiant(e)s qui ont abordé les codes concernant ce thème. Et la portée des variables est un thème qui est aussi bien compris.

Question 5 – Différences [14 pts]

Voici, sur ces deux pages, trois programmes qui se ressemblent, mais ont certaines différences indiquées en **gras**.
Pour chacun :

1. dites si vous pensez qu'il compile ou non (**indice :** au moins un des trois compile) ;
2. si vous pensez que oui, dites alors ce qu'il affiche ;
3. dans tous les cas, justifiez votre réponse.

Réponses :
`progA.cc` : il compile et affiche :

```
hibou
caillou
chou
genou
```

`y` n'étant ni un pointeur ni une référence, il y a « résolution statique des liens », c.-à-d. appel direct des méthodes de `Truc`.
`progB.cc` : Non il ne compile pas pour plusieurs raisons liées aux droits d'accès qui empêchent les lignes 47 à 49.
`progC.cc` : il compile et affiche :

```
hibou
caillou
Festival
genou
```

C'est presque comme `progA`, mais `y` étant ici un pointeur il peut y avoir « résolution dynamique des liens » (polymorphisme) sur les méthode virtuelles (`c()` et `d()`, donc ; rien ne change pour `a()` et `b()`).
Cependant (piège !), la méthode `Blob::d()` n'a pas la même signature que celle de `Truc` et n'est donc pas une redéfinition de celle-ci. `Truc::d()` n'étant pas redéfinie, c'est bien elle (la seule) qui est appelée.

Commentaires de correction : globalement réussie à 63.0%. Les principales erreurs ont été :

- trouver du polymorphisme dans `progA` ;
- penser que `progA` ne compile pas parce que le constructeur de copie par défaut ne serait présent à cause de la présence du constructeur écrit (ce n'est *que* le constructeur *par défaut* par défaut (2 fois) qui est supprimé) ;
- tomber dans le piège du `progC` ; trop peu d'étudiant(e)s remarquent que la méthode `Blob::d()` dans a une signature différente de celle de `Truc`

`progA.cc`

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Truc {
 5  public:
 6    void a() const {
 7      cout << "hibou" << endl;
 8    }
 9
10    void b() const {
11      cout << "caillou" << endl;
12    }
13
14    virtual void c() const {
15      cout << "chou" << endl;
16    }
17
18    virtual void d() const {
19      cout << "genou" << endl;
20    }
21  };
22
23  class Blob : public Truc {
24  public:
25    void a() const {
26      cout << "Bal" << endl;
27    }
28
29    void b() const {
30      cout << "Chacal" << endl;
31    }
32
33    void c() const override {
34      cout << "Festival" << endl;
35    }
36
37    void d(double x = 0.0) {
38      cout << "Blob::d(" << x
39           << ")" << endl;       }
40  };
41
42  int main()
43  {
44    Blob x;
45    Truc y(x);
46    y.a();
47    y.b();
48    y.c();
49    y.d();
50    return 0;
51  }
```

progB.cc

```
1  #include <iostream>
2  using namespace std;
3
4  class Truc {
5  public:
6    void a() const {
7      cout << "hibou" << endl;
8    }
9  protected:
10   void b() const {
11     cout << "caillou" << endl;
12   }
13
14   virtual void c() const {
15     cout << "chou" << endl;
16   }
17 private:
18   virtual void d() const {
19     cout << "genou" << endl;
20   }
21 };
22
23 class Blob : public Truc {
24 public:
25   void a() const {
26     cout << "Bal" << endl;
27   }
28
29   void b() const {
30     cout << "Chacal" << endl;
31   }
32
33   void c() const override {
34     cout << "Festival" << endl;
35   }
36
37   void d(double x = 0.0) {
38     cout << "Blob::d(" << x
39          << ")" << endl;      }
40 };
41
42 int main()
43 {
44   Blob x;
45   Truc y(x);
46   y.a();
47   y.b();
48   y.c();
49   y.d();
50   return 0;
51 }
```

progC.cc

```
1  #include <iostream>
2  using namespace std;
3
4  class Truc {
5  public:
6    void a() const {
7      cout << "hibou" << endl;
8    }
9
10   void b() const {
11     cout << "caillou" << endl;
12   }
13
14   virtual void c() const {
15     cout << "chou" << endl;
16   }
17
18   virtual void d() const {
19     cout << "genou" << endl;
20   }
21 };
22
23 class Blob : public Truc {
24 public:
25   void a() const {
26     cout << "Bal" << endl;
27   }
28
29   void b() const {
30     cout << "Chacal" << endl;
31   }
32
33   void c() const override {
34     cout << "Festival" << endl;
35   }
36
37   void d(double x = 0.0) {
38     cout << "Blob::d(" << x
39          << ")" << endl;      }
40 };
41
42 int main()
43 {
44   Blob x;
45   Truc* y(&x);
46   y->a();
47   y->b();
48   y->c();
49   y->d();
50   return 0;
51 }
```