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 Système, sections IN et SC
Chappelier J.-C.

Programmation Orientée Objet (SMA/SPH) :

CORRIGÉ DE LA SÉRIE NOTÉE

28 avril 2022

INSTRUCTIONS (à lire attentivement)

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

1. Vous disposez de 1h45 pour faire cette série notée (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 ; utilisez aussi le verso des feuilles, **MAIS** n'utilisez *que* le verso de la feuille sur laquelle se trouve la question, et non **pas** celui de la feuille précédente !
   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, ou si vous avez un doute, demandez des précisions à l'un(e) des assistant(e)s.
6. Cette série notée comporte deux exercices indépendants (pages 2 et 6), 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 95). Les deux exercices comptent pour la note finale.

Exercice 1 – SU(2) [sur 30 points]

Nous nous intéressons dans cet exercice à une représentation matricielle spécifique d'éléments d'un ensemble répondant au doux nom de « groupe de Lie SU(2) ». Dans cette représentation, un élément de SU(2) est une matrice de la forme :

$$\begin{pmatrix} a & b \\ -\operatorname{conj}(b) & \operatorname{conj}(a) \end{pmatrix}$$

telle que son déterminant vaille 1, c.-à-d.

$$\begin{vmatrix} a & b \\ -\operatorname{conj}(b) & \operatorname{conj}(a) \end{vmatrix} = |a|^2 + |b|^2 = 1,$$

où a et b sont des nombres complexes ; $\operatorname{conj}(z)$ et $|z|$ représentant respectivement le nombre complexe conjugué et le module du nombre complexe z.

Rappelons que si un nombre complexe z est représenté par le couple de réels (x, y) tels que $z = x + iy$, son conjugué $\operatorname{conj}(z)$ est alors représenté par $(x, -y)$ et le carré de son module (c.-à-d. $|z|^2$) vaut x^2+y^2.

La somme de deux éléments P et Q de SU(2), tels que

$$P = \begin{pmatrix} a & b \\ -\operatorname{conj}(b) & \operatorname{conj}(a) \end{pmatrix} \quad \text{et} \quad Q = \begin{pmatrix} c & d \\ -\operatorname{conj}(d) & \operatorname{conj}(c) \end{pmatrix},$$

est donnée par :

$$P + Q = \begin{pmatrix} a+c & b+d \\ -\operatorname{conj}(b+d) & \operatorname{conj}(a+c) \end{pmatrix},$$

et leur produit est donné par :

$$P \times Q = \begin{pmatrix} a \cdot \operatorname{conj}(c) + b \cdot \operatorname{conj}(d) & -a \cdot d + b \cdot c \\ -\operatorname{conj}(-a \cdot d + b \cdot c) & \operatorname{conj}(a \cdot \operatorname{conj}(c) + b \cdot \operatorname{conj}(d)) \end{pmatrix}.$$

Le but de cet exercice est de réaliser une implémentation C++ (orientée-objet) permettant la manipulation de cette représentation des éléments de SU(2).

Pour la représentation des nombres complexes, on utilisera celle fournie par le langage C++. On supposera que l'extrait de code suivant existe déjà :

```
#include <complex>
using namespace std;

typedef complex<double> Complexe; // pour simplifier
```

ce qui permettrait *par exemple* des utilisations comme suit :

```
Complexe i(0,1); // exemple de construction : l'imaginaire pur
Complexe a, b, c;    // trois complexes, pour l'exemple
...
a = b * c;                    // exemple de multiplication
Complexe conjugue = conj(a);  // exemple de calcul du conjugué
```

```
double module = a.abs();     // exemple de calcul du module
cout << a;                   // exemple d'affichage
if (a != b) { ...            // exemple de comparaison
```

Question 1.1 – Type `MatriceSU2` [sur 8 points]

[3 points] Définissez un type `MatriceSU2` permettant de représenter en C++ les éléments de SU(2) tels que définis précédemment.

[5 points] Définissez également ce qui est nécessaire pour pouvoir les initialiser (libre à vous de faire vos choix).

Nous **ne** vous demandons **pas** d'anticiper ici les questions suivantes. Si vos réponses aux questions suivantes devaient ajouter quelque chose à ce que vous avez écrit ici, nous supposerons que ce serait fait de façon correcte.

```
class MatriceSU2
{
public:
  MatriceSU2(Complexe a_ = Complexe(1,0), Complexe b_ = Complexe(0,0))
    : a(a_), b(b_)
  {
    // do something about non-unray determinant; whatever
    const double det(norm(a) + norm(b));
    if (abs(det) < 1e-12) throw; // or whatever
    else {
      // or whatever (not required)
      a /= sqrt(det);
      b /= sqrt(det);
    }
  }
private:
  Complexe a;
  Complexe b;
};
```

Un point que nous voulions évaluer ici est la non redondance de l'information (distinction entre implémentation et « interface » (au sens large)).

Nous n'attendions pas la formule spécifique pour la normalisation du déterminant, simplement de prévoir quelque chose si le déterminant n'est pas 1.

Question 1.2 – Affichage [sur 5 points]

Définissez l'opérateur d'affichage `<<` pour vos `MatriceSU2`. Nous vous laissons libre choix du format de cet affichage (pas nécessaire de mettre en forme de façon compliquée).

Si vous ajoutez quelque chose au type `MatriceSU2`, il **n'**est **pas** nécessaire de modifier la question précédente. Donnez simplement ici la/les *définition(s)* (pas les prototypes) de tout ce qu'il vous semble nécessaire d'ajouter et que vous utilisez ici. Vous *pouvez* aussi, si nécessaire, compléter votre code C++ par une *brève* explication en français de ce dont il s'agit.

Tout le paragraphe précédent s'applique de façon identique à toutes les question suivantes de cet exercice.

```
void MatriceSU2::serialize(ostream& out) const {
  out << "[ [ " << a << " " << b << " ] [ "
      << -conj(b) << " " << conj(a) << "] ]";
}

ostream& operator<<(ostream& out, MatriceSU2 const & M) {
  M.serialize(out);
  return out;
}
```

Comme nous ne demandons que des extraits de code et non pas un programme complet, la définition interne des méthodes (c.-à-d. sans le nom de classe et opérateur de résolution de portée) est aussi tolérée.

Comme il n'y a pas d'affichage polymorphique dans cette question, tout le code pourrait aussi simplement être dans l'`operator<<` ; mais il faudrait alors, soit des accesseurs (le dire), soit un `friend` (toléré ici).

Question 1.3 – Comparaisons [sur 7 points]

Ajoutez les opérateurs de comparaison `==` et `!=`.

P et Q tels que définis dans l'introduction sont dits égaux si et seulement si $a = c$ et $b = d$.

```
bool MatriceSU2::operator==(MatriceSU2 const & autre) const {
  return (a == autre.a) and (b == autre.b);
}


bool MatriceSU2::operator!=(MatriceSU2 const & autre) const {
  return not (*this == autre);
}
```

Question 1.4 – Addition et soustraction [sur 5 points]

Définissez les opérateurs d'addition (`+`) et de soustraction (`-`) pour les éléments de SU(2).

Remarque importante : on ne demande que les opérateurs combinant deux matrices (c.-à-d. `operator+` et `operator-`), pas les opérateurs d'auto-affectation (c.-à-d. ni `operator+=`, ni `operator-=`) ; vous *pouvez* cependant les définir si cela vous semble utile. Par contre, si vous les utilisez ici, alors vous *devez* les définir ici.

```
MatriceSU2 MatriceSU2::operator+(MatriceSU2 const & autre) const {
  return MatriceSU2(a + autre.a, b + autre.b);
}

MatriceSU2 MatriceSU2::operator-(MatriceSU2 const & autre) const {
  // Remarque : on pourrait aussi fournir l'opposé (operator-())
  return MatriceSU2(a - autre.a, b - autre.b);
}
```

Pour faire simple, nous avons ici fait une surchage *interne* de ces opérateurs (ce qui est tout à fait possible). Si l'on préfère la surcharge externe, il faudrait alors, soit des getter (bof), soit, en effet définir les opérateurs d'auto-affectation.

Par aillers, il faut ici absolument éviter de dupliquer du code (plus qu'une ligne). Avec une solution plus complexe, il est donc préférable de fournir l'opérateur « opposé » (`operator-()` sans argument).

Remarque mathématique : à noter que SU(2) n'est pas un groupe additif, donc en toute rigueur l'addition est mal définie ; la gestion de ces cas n'était pas attendue (dans la solution proposée, ils sont gérés via le constructeur).

Question 1.5 – Multiplication [sur 5 points]

Définissez l'opérateur de multiplication (`*`) pour les éléments de SU(2).

Remarque : on ne demande que l'opérateur combinant deux matrices (c.-à-d. `operator*`), pas l'opérateur d'auto-affectation (c.-à-d. pas `operator*=`), mais vous *pouvez* cependant le définir si cela vous semble utile. Par contre, si vous l'utilisez ici, alors vous *devez* le définir ici.

```
MatriceSU2 MatriceSU2::operator*(MatriceSU2 const & autre) const {
  return MatriceSU2(  a * conj(autre.a) + b * conj(autre.b) ,
                     -a *      autre.b  + b *      autre.a  );
}
```

(Remarque mathématique : ce n'est pas le produit usuel des matrices, mais un produit hermitien : $P \times Q = P\,\overline{Q}^t$)

Exercice 2 – Employés d'une société informatique [sur 65 points]

On cherche dans cet exercice à écrire un programme, en conception orientée-objet *évitant toute duplication de code*, qui permette de gérer le salaire d'employés d'une entreprise d'informatique.

Les employés que l'on souhaite représenter sont caractérisés chacun par un nom (qui ne changera pas une fois donné), un revenu mensuel et un taux d'occupation (pourcentage de temps travaillé par mois ; par exemple : emploi à 80%).

Il existe trois catégories d'employés :

- les managers, qui ont en plus un nombre de jours voyagés et un nombre de nouveaux clients apportés ;
- les testeurs, qui ont un nombre d'erreurs corrigées ;
- les programmeurs, qui ont un nombre de projets achevés.

Question 2.1 – Types de données [sur 21 points]

On vous demande de définir ici *tous* les types de données qui vous semblent nécessaires à la modélisation de la description précédente.

Contrairement à l'Exercice 1, nous vous demandons ici d'être exhaustifs et d'inclure tous les *prototypes* nécessaires à toute la suite, mais **uniquement** les prototypes des méthodes, pas leurs définitions qui sont demandées par la suite. Nous vous encourageons donc à déjà lire toutes les questions suivantes, ou alors à prévoir de la place dans votre réponse pour la compléter plus tard.

Il faut aussi, bien sûr, mettre ici les attributs. Nous **ne** vous demandons par contre **pas** de fournir de « méthode `set` », ni de « méthode `get` », ni de surcharger l'opérateur `<<`.

```
class Employe {
public:
  Employe(string const& nom, double revenu, unsigned int un = taux_par_defaut);
  virtual ~Employe();

  // bonus (avancé)
  Employe(Employe const&)            = default;
  Employe& operator=(Employe const&) = default;
  Employe(Employe&&)                 = default; // encore + avancé
  Employe& operator=(Employe&&)      = default; // encore + avancé

  virtual void afficher() const;
  virtual double revenu_annuel() const;

protected:
  static constexpr double taux_par_defaut = 100;

private:
  const string nom;
  double revenu;
  unsigned int taux;
};
```

Note : le taux peut bien sûr être `double`.

Une version plus aboutie (voir Question 2.5) pourrait nécessiter une méthode de copie polymorphique.

Bonus :

- Ayant mis un destructeur virtuel, certain(e)s, plus avancé(e)s, penseront à rajouter le constructeur de copie et l'opérateur d'affectation ; voire, encore plus avancé(e)s, aussi pour le déplacement.
- Le `taux_par_defaut`, `protected` afin de pouvoir le partager aux sous-classes, est un bonus (avancé) pour éviter la copie de cette valeur dans tous les constructeurs.

```
class Manager : public Employe {
public:
  Manager(string const& nom, double revenu, unsigned int jours, unsigned int clients,
          unsigned int taux = taux_par_defaut);

  // [...] comme pour Employe


private:
  unsigned int jours_voyages;
  unsigned int nb_clients;
  static constexpr double francs_par_client = 500.0;
  static constexpr double francs_par_jour   = 100.0;
};

class Testeur : public Employe {
public:
  Testeur(string const& nom, double revenu, unsigned int erreurs
          , unsigned  int taux = taux_par_defaut);

  // [...] comme pour Employe


private:
  unsigned int erreurs; // + éventuelle constante de classe
};

class Programmeur : public Employe {
  public:
  Programmeur(string const& nom, double revenu, unsigned int projets,
              unsigned int taux = taux_par_defaut);

  // [...] comme pour Employe


private:
  unsigned int projets; // + éventuelle constante de classe
};
```

Question 2.2 – Initialisations [sur 7 points]

Dotez chacun de vos types d'un moyen permettant d'initialiser *toutes* les valeurs des attributs concernés et tel que le taux d'occupation soit 100% par défaut.

Si une initialisation reçoit un taux d'occupation inférieur à 10%, le taux effectivement retenu doit être de 10%. De même, si le taux d'occupation reçu est supérieur à 100%, il devra être limité à 100%.

Notez ici (et au dos) les *définitions* nécessaires à votre réponse, et ajoutez les prototypes à votre réponse à la Question 2.1.

```
Employe::Employe(string const & un_nom, double un_revenu, unsigned int un_taux)
  : nom(un_nom), revenu(un_revenu), taux(un_taux)
{
  if (taux < 10)       taux =  10;
  else if (taux > 100) taux = 100;
}

Manager::Manager(string const& un_nom, double un_revenu, unsigned int nb_jours_voyages,
                 unsigned int nouveaux_clients, unsigned int un_taux)
  : Employe(un_nom, un_revenu, un_taux), jours_voyages(nb_jours_voyages)
  , nb_clients(nouveaux_clients)
{}

Testeur::Testeur(string const& un_nom, double un_revenu, unsigned int nb_erreurs,
                 unsigned int un_taux)
  : Employe(un_nom, un_revenu, un_taux), erreurs(nb_erreurs)
{}

Programmeur::Programmeur(string const& un_nom, double un_revenu, unsigned int nb_projets,
                         unsigned int un_taux)
  : Employe(un_nom, un_revenu, un_taux), projets(nb_projets)
{}
```

Question 2.3 – Revenu annuel [sur 8 points]

Ajoutez ensuite un moyen de calculer, pour chaque employé, *son* revenu annuel comme suit :

- tout employé a un revenu de base qui vaut 12 fois [1] son revenu mensuel multiplié par son taux d'occupation ;
- pour un manager, on ajoute un bonus de 500 francs pour chaque client apporté, et de 100 francs pour chaque jour voyagé ;
- pour un testeur, on ajoute un bonus de 10 francs pour chaque erreur corrigée ;
- et pour un programmeur, on ajoute un bonus de 200 francs pour chaque projet achevé.

Notez ici les *définitions* nécessaires à votre réponse, et ajoutez les prototypes à votre réponse à la Question 2.1.

1. car il y a douze mois dans l'année.

```
double Employe::revenu_annuel() const {
  // these are NOT magic numbers: 12 months and 100%
  return revenu * 12.0 * taux / 100.0;                }

double Manager::revenu_annuel() const {
  return Employe::revenu_annuel() + nb_clients * francs_par_client
    + jours_voyages * francs_par_jour;
}

double Testeur::revenu_annuel() const {
  return Employe::revenu_annuel() + erreurs * 10; // mieux : constante de classe
}

double Programmeur::revenu_annuel() const {
  return Employe::revenu_annuel() + projets * 200; // mieux : constante de classe
}
```

Question 2.4 – Fin d'instance [sur 4 points]

À chaque fois qu'une instance d'employé est détruite, nous souhaiterions qu'un message soit affiché, au format :
`Nous cherchons un(e) <catégorie> : <nom> a quitté l'entreprise.`
où « `<catégorie>` » représente la catégorie de l'employé (manager, programmeur ou testeur) et « `<nom>` », son nom ; par exemple :
`Nous cherchons un(e) manager : Caroline Legrand a quitté l'entreprise.`
`Nous cherchons un(e) programmeur : Paul Lepetit a quitté l'entreprise.`
`Nous cherchons un(e) testeur : Pierre Lelong  a quitté l'entreprise.`
Définissez ici ce qui est nécessaire, et ajoutez si nécessaire les prototypes à votre réponse à la Question 2.1.

```
Employe::~Employe() {
  cout << nom << " a quitté l'entreprise." << endl;
}

Manager::~Manager() {
  cout << "Nous cherchons un(e) manager : ";
}

Testeur::~Testeur() {
  cout << "Nous cherchons un(e) testeur : ";
}

Programmeur::~Programmeur() {
  cout << "Nous cherchons un(e) programmeur : ";
}
```

Question 2.5 – Entreprise [sur 7 points]

Définissez ici complètement le type `Entreprise`, simplement comme une liste d'employés (initialement vide). On veut pouvoir ajouter des employés à cette liste au moyen de la méthode `embauche()`.

On supposera que le programme n'utilise pas les employés autrement que dans l'`Entreprise`.

Il y a là plusieurs solutions possibles en fonction des choix effectués pour la collection hétérogène.

La version la plus simple :

```
class Entreprise
{
public:
  void embauche(Employe* p) {
    staff.push_back(p);
  }

private:
  vector<Employe*> staff;
};
```

La version plus avancée conforme au cours :

```
class Entreprise
{
public:
  void embauche(Employe const& p) {
    staff.push_back(p.copie());
  }

private:
  vector<unique_ptr<Employe>> staff;
};
```

mais cette solution nécessaire la définition de la copie polymorphique des `Employes` :

— méthode virtuelle pure dans la super-classe :

```
  virtual unique_ptr<Employe> copie() const = 0;
```

— et définition dans les sous-classes :

```
unique_ptr<Employe> Manager::copie() const
{ return unique_ptr<Employe>(new Manager(*this)); }

unique_ptr<Employe> Testeur::copie() const
{ return unique_ptr<Employe>(new Testeur(*this)); }

unique_ptr<Employe> Programmeur::copie() const
{ return unique_ptr<Employe>(new Programmeur(*this)); }
```

Une version intermédiaire entre les deux précédentes :

```
class Entreprise
{
public:
  void embauche(Employe* p) {
    staff.push_back(unique_ptr<Employe>(p));
  }

private:
  vector<unique_ptr<Employe>> staff;
};
```

Et [non attendu, mais correct tout de même] une version encore plus avancée (sans copie polymorphique, mais avec déplacement) :

```
class Entreprise
{
public:
  void embauche(unique_ptr<Employe> && p) {
    staff.push_back(move(p));
  }

private:
  vector<unique_ptr<Employe>> staff;
};
```

Question 2.6 – Exemple de `main()` [sur 5 points]

En supposant que les `Entreprise`s ont une méthode `afficher()` (que nous **ne** vous demandons **pas** d'écrire) qui affiche chacun de ses employés, complétez le `main()` suivant :

[...]

pour qu'il affiche (à l'ordre des employés près) :

[...]

Il y a là plusieurs solutions possibles en fonction des choix effectués pour la collection hétérogène `Entreprise`.

Avec la version la plus simple :

```
Manager     employe1("Caroline Legrand", whatever_1,  30,  4 );
Programmeur employe2("Paul Lepetit" ,    whatever_2,   3     );
Testeur     employe3("Pierre Lelong",    whatever_3, 124, 50 );

e.embauche(&employe1);
e.embauche(&employe2);
e.embauche(&employe3);
```

Avec la version plus avancée avec copie polymorphique :

```
e.embauche(Manager    ("Caroline Legrand", whatever_1,  30,  4 ));
e.embauche(Programmeur("Paul Lepetit" ,    whatever_2,   3     ));
e.embauche(Testeur    ("Pierre Lelong",    whatever_3, 124, 50 ));
```

Avec la version intermédiaire :

```
e.embauche(new Manager    ("Caroline Legrand", whatever_1,  30,  4 ));
e.embauche(new Programmeur("Paul Lepetit" ,    whatever_2,   3     ));
e.embauche(new Testeur    ("Pierre Lelong",    whatever_3, 124, 50 ));
```

Avec la version encore plus avancée (déplacement) :

```
e.embauche(make_unique<Manager>    ("Caroline Legrand", whatever_1,  30,  4 ));
e.embauche(make_unique<Programmeur>("Paul Lepetit" ,    whatever_2,   3     ));
e.embauche(make_unique<Testeur>    ("Pierre Lelong",    whatever_3, 124, 50 ));
```

Question 2.7 – Affichage [sur 13 points]

En supposant que les `Entreprises` ont une méthode `afficher()` (que nous **ne** vous demandons **pas** d'écrire) qui appelle simplement la méthode « `afficher()` » de chacun de ses employés, définissez ici une telle méthode « `afficher()` » (pour chaque catégorie d'employés) et tout ce qui est nécessaire de façon à ce que le `main()` précédent (correctement complété) affiche ce qui est donné.

```
void Employe::afficher() const {
    cout << nom << " :" << endl;
    cout << "  Taux d'occupation : " << taux << "%. ";
    cout << "Salaire annuel : " << revenu_annuel() << " francs." << endl;
}

void Manager::afficher() const {
  cout << "Manager ";
  Employe::afficher();
  cout << "  A voyagé " << jours_voyages << " jours et apporté "
       << nb_clients << " nouveaux clients." << endl;
}

void Testeur::afficher() const {
  cout << "Testeur ";
  Employe::afficher();
  cout << "  A corrigé " << erreurs << " erreurs." << endl;
}

void Programmeur::afficher() const {
  cout << "Programmeur ";
  Employe::afficher();
  cout << "  A mené à bien " << projets << " projets." << endl;
}
```

Note : les messages dus aux destructeurs ont dû être traités en Question 2.4.