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

18 avril 2024

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 de la place supplémentaire en pages 10 et 11.

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 ne comporte qu'un seul exercice (en 4 questions ; total : 96 points).

Exercice 1 – Diplomatie

Cadre général

On s'intéresse ici à simuler un jeu de diplomatie constitué de différents représentants [1] essayant de se convaincre les uns les autres.

On vous demande pour cela d'écrire des portions d'un programme en C++, en utilisant une approche « orientée objets » avec le moins de duplication de code possible et sans aucune fuite de mémoire.

Pour déjà donner une idée générale, voici deux exemples de `main()` possibles et le résultat attendu correspondant (le même pour les deux). Regardez les bien avant de commencer à coder et choissez votre version. N'hésitez pas à y revenir y autant que nécessaire lors de la lecture de la suite de la donnée. Pour que ce soit plus pratique, ces exemples sont redonnés en toute dernière dernière page.

```
// VERSION 1 (au choix)

int main()
{
  Assembly a;
  a.greet(new Expert(1));
  a.greet(new Novice(1));
  a.greet(new Expert(2));
  a.greet(new Novice(2));

  cout << a << endl;

  a.meet(0, 1);
  a.meet(0, 3);
  a.meet(0, 2);

  cout << a << endl;

  return 0;
}
```

```
// VERSION 2 (au choix)

int main()
{
  Assembly a;
  a.greet(make_unique<Expert>(1));
  a.greet(make_unique<Novice>(1));
  a.greet(make_unique<Expert>(2));
  a.greet(make_unique<Novice>(2));

  cout << a << endl;

  a.meet(0, 1);
  a.meet(0, 3);
  a.meet(0, 2);

  cout << a << endl;

  return 0;
}
```

```
The assembly is made up of:
an expert, resolution: 1, charisma: 6
a novice, resolution: 1, charisma: 3
an expert, resolution: 2, charisma: 6
a novice, resolution: 2, charisma: 3

an expert meets a novice (supporter):
an expert congratulates
a novice listens diligently

an expert meets a novice (opponent):
an expert argues for resolution 1
a novice gets convinced
```

Suite :

```
an expert meets an expert (opponent):
an expert argues for resolution 1
an expert tries to convince

The assembly is made up of:
an expert, resolution: 1, charisma: 7
a novice, resolution: 1, charisma: 3
an expert, resolution: 2, charisma: 6
a novice, resolution: 1, charisma: 3
```

Notez comme le charisme du premier expert et la résolution défendue par le dernier diplomate ont changé entre le premier et le dernier paragraphe (il a été amadoué [2]).

1. "*representatives*" in English.
2. "*convinced*" in English.

On aura donc une assemblée[3] constituée de représentants. Pour simplifier, nous n'aurons que deux sortes de représentants : les experts et les novices.

Chaque représentant aura une résolution à défendre (un simple entier positif, ici, pour simplifier) et un niveau de charisme (aussi un entier positif).

Les représentants pourront se rencontrer[4] pour essayer de convaincre l'autre d'adopter sa résolution. Ils le feront au travers de dialogues expliqués plus loin plus en détails.

L'assemblée, ainsi que n'importe quel représentant, devront pouvoir être affichés en utilisant l'opérateur usuel de C++ (l'opérateur <<).

Question 1.1 – Les représentants [sur 40 points]

Commencez par créer la ou les classes nécessaires à la représentation des représentants.

Les représentants novices auront un charisme de 3 par défaut et les représentants experts un charisme de 6 par défaut.

Ajoutez une méthode permettant de connaître la résolution défendue par le représentant. Pour simplifier, on ne fera pas ici de « méthode *get* » pour le charisme, ni aucune « méthode *set* ».

On souhaiterait par ailleurs pouvoir augmenter de 1 le charisme d'un représentant en utilisant simplement l'opérateur `++` (sur le représentant).

Ajoutez aussi une méthode `whoami()`, qui, reçoit un flot en paramètre et, pour les représentants experts affiche « `an expert` », pour les représentants novices affiche « `a novice` ».

Et n'oubliez pas que les représentants doivent pouvoir être affichés en utilisant l'opérateur usuel (<<), au format donné dans l'exemple de départ.

Réponse : (vous pouvez répondre en deux colonnes si nécessaire, et avez aussi de place au dos.)

La première chose à faire est d'éviter le copié-collé de l'affichage en définissant une classe pour cela :

```
class Printable {
public:
  virtual void display(ostream& out) const  = 0;
  virtual ~Printable() = default;
  // optional: rule of 3 (or 5)
};

ostream& operator<<(ostream& out, const Printable& something)
{
  something.display(out);
  return out;
}
```

3. "*assembly*" in English.
4. "*meet*" in English.

Ensuite, les représentants peuvent s'organiser comme suit :

```
class Representative : public Printable {
public:
  Representative(unsigned int r, unsigned int c)
    : resolution_(r), charisma_(c)
  {}

  virtual ~Representative() = default;

  virtual void whoami(ostream& out) const = 0;

  void display(ostream& out) const override {
    whoami(out);
    out << ", resolution: " << resolution_;
    out << ", charisma: "   << charisma_  ;
  }

  unsigned int resolution() const { return resolution_; };

  Representative& operator++() { ++charisma_; return *this; }
  // ou aussi : void operator++() { ++charisma_; }

private:
  unsigned int resolution_;
  unsigned int charisma_;
};

class Novice : public Representative {
public:
  Novice(unsigned int r, unsigned int c = 3) : Representative(r, c) {}

  void whoami(ostream& out) const override
  { out << "a novice"; }
};

class Expert : public Representative {
public:
  Expert(unsigned int r, unsigned int c = 6) : Representative(r, c) {}

  void whoami(ostream& out) const override
  { out << "an expert"; }
};
```

Question 1.2 – L'assemblée [sur 15 points]

Commencez par indiquer laquelle vous choisissez parmi les deux versions de `main()` données en exemple au départ.

Définissez ensuite une classe `Assembly`, qui représente une assemblée comme une liste de représentants (libre à vous de choisir le type exact). Cette classe devra avoir une méthode `greet()` qui permet d'ajouter un représentant (libre à vous de choisir le type exact).

Conseil : ne pas faire trop compliqué ici ; au niveau de ce devoir, nous n'attendons *aucune* copie, encore moins polymorphique.

N'oubliez pas que l'assemblée doit pouvoir être affichée en utilisant l'opérateur usuel, au format donné dans l'exemple de départ. Nous rappelons aussi que le programme complet ne doit avoir aucune fuite de mémoire (sans aucune modification du `main()` que vous avez choisi).

Réponse :

Version 1a :

```
class Assembly : public Printable {
public:
  ~Assembly() {
    for (auto x : people) {
      delete x;
    }
  }

  void greet(Representative* somebody) {
    if (somebody != nullptr) people.push_back(somebody);
  }

  void display(ostream& out) const override {
    cout << "The assembly is made up of:" << endl;
    for (auto x : people) {
      out << *x << endl;
    }
  }

private:
  vector<Representative*> people;
};
```

Version 1b :

```
class Assembly : public Printable {
public:
  void greet(Representative* somebody) {
    if (somebody != nullptr)
      people.push_back(unique_ptr<Representative>(somebody));
  }

  void display(ostream& out) const override {
    cout << "The assembly is made up of:" << endl;
    for (auto const& x : people) {
      out << *x << endl;
    }
  }

private:
  vector<unique_ptr<Representative>> people;
};
```

Version 2 :

```
class Assembly : public Printable {
public:
  // allowed: void greet(unique_ptr<Representative> somebody) {
  void greet(unique_ptr<Representative> && somebody) {
    people.push_back(move(somebody));
  }

  void display(ostream& out) const override {
    cout << "The assembly is made up of:" << endl;
    for (auto const& v : people) {
      out << *v << endl;
    }
  }

private:
  vector<unique_ptr<Representative>> people;
};
```

Question 1.3 – Rencontre entre représentants [sur 7 points]

On suppose que tous les représentants ont une méthode `dialogue()` qui prend un représentant en paramètre (ce sera l'objet de la question suivante).

Définissez ici une méthode `meet()` de la classe `Assembly` qui permet de faire se rencontrer deux de ses représentants. Cette méthode reçoit deux index (positions de deux représentants dans la liste) et (revoir l'exemple de départ si nécessaire) :

1. affiche la rencontre comme par exemples :

   ```
   an expert meets a novice
   an expert meets an expert
   a novice meets a novice
   a novice meets an expert
   ```

2. si les résolutions des deux représentants sont égales, affiche ensuite « `(supporter)` », et sinon affiche « `(opponent)` », suivi de ':' et d'un retour à la ligne ;
3. appelle la méthode `dialogue()` du premier représentant en passant en argument le second représentant.

Réponse :

```
void meet(size_t i, size_t j) {
  if (i >= people.size()) // some error handling, e.g. throw
  if (j >= people.size()) // we can also use at[i] instead of [i]

  people[i]->whoami(cout);
  cout << " meets ";
  people[j]->whoami(cout);

  if (people[i]->resolution() == people[j]->resolution()) {
    cout << " (supporter)";
  } else {
    cout << " (opponent)";
  }

  cout << ":" << endl;
  people[i]->dialogue(*people[j]);
  cout << endl;
}
```

Question 1.4 – Dialogues entre représentants [sur 34 points]

Tous les représentants devront maintenant avoir quatre méthodes supplémentaires : `dialogue()`, `speak_to()`, `reply()` et `is_convinced_by()`. Toutes ces méthodes ne retournent rien et reçoivent un représentant comme paramètre.
Seule la méthode `speak_to()` ne modifie ni l'instance courante, ni son paramètre. Les trois autres sont susceptibles de modifier l'un ou l'autre (ou les deux).

La méthode `is_convinced_by()` est la même pour tous les représentants : si le charisme du représentant reçu en paramètre est strictement supérieur à celui de l'instance courante alors
- la résolution de l'instance courante devient celle du représentant reçu en paramètre ;
- on appelle l'opérateur `++` sur le représentant reçu en paramètre

(et sinon, il ne se passe rien).

La méthode `dialogue()` est aussi la même pour tous les représentants : elle doit

1. appeller la méthode `whoami()` de l'instance courante ;
2. passer son paramètre (représentant) à la méthode `speak_to()` de l'instance courante (on parle à l'autre) ;
3. appeller la méthode `whoami()` du représentant reçu en paramètre ;
4. faire appel à la méthode `reply()` du représentant reçu en paramètre en lui passant l'instance courante comme argument (l'autre nous répond).

Le comportement des méthodes `speak_to()` et `reply()` dépend du type de représentant. Pour simplifier, pour toutes les actions décrites ci-dessous sauf `is_convinced_by()`, il suffira simplement d'afficher un bref message indiquant l'action effectuée (revoir si nécessaire l'exemple de départ).

`speak_to()` :
- si un représentant novice parle à un représentant partisan [5] (c.-à-d. défendant la même résolution), il dit simplement bonjour, sinon il discute [6] de sa résolution ;
- si un représentant expert parle à un représentant partisan, il le congratule [7], sinon il défend [8] sa résolution.

`reply()` :
- un représentant novice écoute religieusement [9] un représentant partisan, et se laisse convaincre (méthode `is_convinced_by()`) par un représentant opposant ;
- un représentant expert exprime son accord [10] à un représentant partisan, et essaye de convaincre [11] un représentant opposant (affichage, puis méthode `is_convinced_by()`, voir l'exemple de déroulement).

Définissez toutes les méthodes nécessaires au fonctionnement décrit ci-dessus, **en indiquant à chaque fois dans quelle classe vous définissez la méthode et avec quels droits d'accès**.

Réponse :

5. "*supporter*" in English. 6. "*discuss*" in English. 7. "*congratulates*" in English. 8. "*argues for*" in English. 9. "*listens diligently*" in English. 10. "*agrees*" in English. 11. "*tries to convince*" in English.

Pour `dialogue()` et `is_convinced_by()`, cela se fait au niveau des `Representatives`, en `public` bien sûr :

```
void dialogue(Representative& other) {
  whoami(cout);
  speak_to(other);
  other.whoami(cout);
  other.reply(*this);
}

void is_convinced_by(Representative& other) {
  if ( other.charisma_ > charisma_ ) {
    resolution_ = other.resolution_;
    ++other;
  }
}
```

Note : `is_convinced_by()`, ne peut pas être que `protected` en raison de `Expert::reply()` qui en a besoin comme `public` pour l'appel sur « l'autre ».

Pour `speak_to()` et `reply()`, il faut mettre en place le polymorphise au niveau des `Representatives` ; cela peut se faire en `private` :

```
private: // or protected, but not public
  virtual void speak_to(Representative const&) const = 0;
  virtual void reply(Representative&) = 0;
```

Il faut ensuite les définir dans chaque sous-classe :

pour les `Novice` :

```
private:
  virtual void speak_to(Representative const& other) const override
  {
    if (other.resolution() == resolution()) {
      cout << " says hello!" << endl;
    } else {
      cout << " discuss her/his resolution (" << resolution() << ")" << endl;
    }
  }

  virtual void reply(Representative& other) override
  {
    if (other.resolution() == resolution()) {
      cout << " listens diligently" << endl;
    } else {
      cout << " gets convinced" << endl;
      is_convinced_by(other);
    }
  }
```

pour les `Expert` :

```
private:
  virtual void speak_to(Representative const& other) const override
  {
    if (other.resolution() != resolution()) {
      cout << " argues for resolution " << resolution() << endl;
    } else {
      cout << " congratulates" << endl;
    }
  }

  virtual void reply(Representative& other) override
  {
    if (other.resolution() != resolution()) {
      cout << " tries to convince" << endl;
      other.is_convinced_by(*this);
    } else {
      cout << " agrees" << endl;
    }
  }
```