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 DU MIDTERM

23 avril 2026

INSTRUCTIONS (à lire attentivement)

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

1. Vous disposez de 1h45 pour faire cet examen (9h15 - 11h00).

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

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

3. Vous avez droit à toute documentation papier.

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

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

   Si nécessaire, il y a de la place supplémentaire en pages 11 et 12.

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

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

6. Cet examen ne comporte qu'un seul exercice (en 11 questions ; total : 87 points).

Exercice 1 – Jolis dessins [87 points]

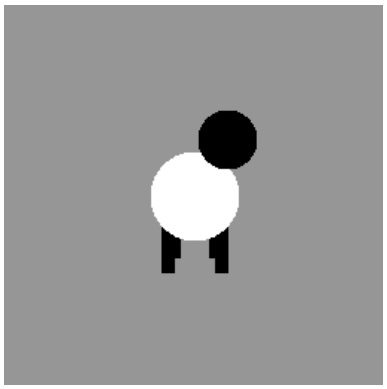
Cadre général

On s'intéresse ici à une façon de créer des images sur un ordinateur. Les images sont ici représentées comme un tableau 2D de couleurs : une couleur pour chacun des pixels (x, y) de l'image.

Pour composer ces images, nous aurons deux abstractions :

- des éléments, ici simplement des cercles et des rectangles (mais plus généralement ce pourrait être n'importe quel composant de base pour dessiner une image) ;
- des scènes, qui sont des collections d'éléments.

Par exemple pour dessiner l'image suivante :



nous avons une « scène mouton », composée de deux éléments cercles et quatre éléments rectangles.
Cette image pourrait être créée avec le `main()` ci-contre [a].

a. sauf qu'imprimé en noir et blanc, le fond vert ressort gris.

```
int main()
{
  const Cercle corps(blanc, 30, 100, 100);
  const Cercle tete (noir , 20, 120,  60);

  const Rectangle patte_1(noir, 5, 20,  80, 130);
  const Rectangle patte_2(noir, 5, 20, 116, 130);
  const Rectangle patte_3(noir, 4, 15,  85, 125);
  const Rectangle patte_4(noir, 4, 15, 111, 125);

  const Scene mouton({
    &patte_1, &patte_2, &patte_3, &patte_4,
    &corps, &tete });

  Image dessin(200, 200, vert);
  dessin << mouton;

  cout << dessin;

  return 0;
}
```

Note : les `Couleur` `blanc`, `noir` et `vert` ont été définies par ailleurs.

Dans la suite, nous allons tour à tour nous intéresser aux couleurs (14.5 points), aux éléments (26.5 points), aux scènes (10.5 points) et enfin aux images (35.5 points).

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

Question 1.1 – Couleurs [14.5 points]

La première étape pour représenter nos images est de représenter les couleurs qui la compose.

Question 1.1.1 – Modèle RVB [5.5 points]

Une façon courante de représenter les couleurs est d'utiliser le modèle dit RVB (rouge, vert, bleu) : une couleur étant alors simplement représentée comme trois entiers (entre 0 et 255, mais peu importe ici) représentant les quantités respectives de rouge, de vert et de bleu.

Définissez ici une classe `Couleur`, avec un constructeur à trois composantes (RVB).
On devra de plus pouvoir lire (mais pas modifier) chacune des composantes (RVB).

```
class Couleur {
private:
    int r, g, b;

public:
    Couleur(int R, int G, int B)
    : r(R), g(G), b(B) {}

    int red()   const { return r; }
    int green() const { return g; }
    int blue()  const { return b; }
};
```

Question 1.1.2 – Teintes de gris [2 points]

Dans le modèle RVB, les teintes de gris pur sont définies par toutes les couleurs ayant une quantité égale de rouge, de vert et de bleu. Cette quantité (égale) s'appelle « niveau de gris ».

Est-il possible d'ajouter à votre classe un constructeur `Couleur(int niveau_de_gris)` ?
Si oui, le définir, si non, expliquer pourquoi.

Oui s'est possible, rien ne s'y oppose.

```
    Couleur(int level)
    : Couleur(level, level, level) {}
```

Commentaire : Beaucoup trop de « copié-collé » ici au lieu de réutiliser le constructeur précédent.

Question 1.1.3 – Modèle CMJ [sur [2 points]

Un autre modèle de couleur possible est le modèle CMJ, utilisé notamment pour les imprimantes. Au lieu d'utiliser rouge, vert et bleu, il utilise cyan, magenta et jaune. On peut passer facilement du modèle CMJ au modèle RVB avec la formule suivante :

$$R = 255 - C \qquad V = 255 - M \qquad B = 255 - J$$

Est-il possible d'ajouter à votre classe un constructeur en CMJ ?
Si oui, le définir, si non, expliquer pourquoi.

Cette question est plus délicate : il n'est pas possible d'ajouter un nouveau constructeur à trois `int` car celui-ci serait ambigu avec la construction RVB.
Par contre (bonus), on peut tout à fait modifier la signature du constructeur CMJ de façon similaire à ce qui a été présenté en cours pour la construction en polaires des complexes ; p.ex. (parmi d'autres) :

```
Couleur(int C, int M, int J, bool useless)
: Couleur(255 - C, 255 - M, 255 - J) {}
```

Question 1.1.4 – Affichage [5 points]

On veut pouvoir utiliser l'opérateur d'affichage `<<` avec nos couleurs, pour les afficher au format `R V B` (avec une espace entre) ; par exemple : `255 128 0`.
On doit pouvoir afficher plusieurs couleurs à la suite, p.ex. :

```
cout << couleur_1 << ' ' << couleur_2 << endl;
```

Définir ici cet opérateur :

```
ostream& operator<<(ostream& sortie, Couleur const& couleur)
{
  sortie << couleur.red() << " " << couleur.green() << " " << couleur.blue();
  return sortie;
}
```

Question 1.2 – Éléments [26.5 points]

Question 1.2.1 – Éléments de base [9.5 points]

Avec les couleurs à disposition, on peut les utiliser pour définir les éléments constitutifs des images : cercles, rectangles, etc.

Un `Element` est défini par une `Couleur` et des coordonnées indiquant son centre (pour simplifier ici, deux `int`, x_c et y_c). Tous ces éléments doivent être fournis à la construction.
On souhaite de plus pouvoir accéder en lecture à la couleur (mais pas en écriture, et aucun accès aux coordonnées du centre).

Pour pouvoir dessiner les éléments, il faut savoir si un point `(x, y)` de l'image (simplement deux `int`) est à l'intérieur on non de cet élément. On impose pour cela une méthode `dedans()` qui prend deux `int`, `x` et `y`, et qui retourne `true` si le point `(x, y)` est dans l'élément et `false` sinon.

Définissez complètement la classe `Element` :

```
class Element {
private:
    Couleur couleur_;

protected:
    int centre_x, centre_y;

public:
    Element(Couleur c, int x, int y)
    : couleur_(c), centre_x(x), centre_y(y) {}

    Couleur couleur() const { return couleur_; }

    virtual bool dedans(int x, int y) const = 0;

    virtual ~Element() = default;
};
```

En version plus avancée :
- les attributs peuvent être `const` ;
- le passage de la `Couleur` au constructeur peut se faire par référence constante ;

— compléter le destructeur virtuel par la règle des trois ou la règle des cinq :

```
Element(Element const&)            = default;
Element& operator=(Element const&) = default;
Element(Element&&)                 = default;
Element& operator=(Element&&)      = default;
```

— ajouter la copie polymorphique (voir Question 1.3) :

```
virtual std::unique_ptr<Element> copie() const = 0;
```

Commentaire : N'oubliez pas le destructeur virtuel !

Limitez au maximum des droits d'accès : `private` tant que possible, `protected` si nécessaire (et jamais d'attribut en `public`)

Question 1.2.2 – Rectangles [8.5 points]

Un rectangle est un élément qui, en plus de son centre (x_c et y_c), a une (demi-)longueur L et une (demi-)hauteur H : un point (x, y) est dans le rectangle si x est compris entre $x_c - L$ et $x_c + L$ et y est compris entre $y_c - H$ et $y_c + H$.

On ne cherche pas à accéder à la longueur ni à la hauteur.

Définir ici l'élément `Rectangle` :[1]

```
class Rectangle : public Element {
private:
    int taille_x, taille_y;

public:
    Rectangle(Couleur c, int L, int H, int x, int y)
    : Element(c, x, y), taille_x(L), taille_y(H) {}

    virtual bool dedans(int x, int y) const override
    {
        return (x + taille_x >= centre_x) and (x <= centre_x + taille_x)
           and (y + taille_y >= centre_y) and (y <= centre_y + taille_y);
    }
};
```

En version plus avancée :

— les attributs peuvent être `const` ;
— les attributs peuvent être `unsigned` ;
— le passage de la `Couleur` au constructeur peut se faire par référence constante ;
— on peut utiliser la valeur absolue `abs()` ;
— on *doit* ajouter la copie polymorphique si elle a été ajoutée précédemment :

```
virtual unique_ptr<Element> Rectangle::copie() const override
{
  return make_unique<Rectangle>(couleur(), taille_x, taille_y, centre_x, centre_y);
}
```

Commentaire : Encore et toujours des

```
if (some_bool) return true; else return false;
```

au lieu de simplement

```
return some_bool;
```

1. Rappel: si nécessaire, voir l'exemple de déroulement fourni en début de donnée (et reproduit en dernière page).

Question 1.2.3 – Cercles [8.5 points]

Un cercle est un élément qui, en plus de son centre (x_c et y_c), a un rayon R : un point (x, y) est dans le cercle si $(x - x_c)^2 + (y - y_c)^2 \leq R^2$.

On ne cherche pas à accéder au rayon.

Définir ici l'élément `Cercle` :[1]

```
class Cercle : public Element {
private:
    int rayon;

public:
    Cercle(Couleur c, int R, int x, int y)
    : Element(c, x, y), rayon(R) {}

    virtual bool dedans(int x, int y) const override
    {
        return pow(x - centre_x, 2) + pow(y - centre_y, 2) <= rayon * rayon;
    }
};
```

En version plus avancée :

- l'attribut peut être `const` ;
- l'attribut peut être `unsigned` ;
- le passage de la `Couleur` au constructeur peut se faire par référence constante ;
- on *doit* ajouter la copie polymorphique si elle a été ajoutée dans `Element`.

Question 1.3 – Scène [10.5 points]

Les éléments de base étant définis, nous pouvons passer à la définition d'outils plus élaborés : les scènes.

Une `Scene` est un ensemble d'`Element`. Cet ensemble est ordonné : on dessinera les éléments dans l'ordre. Si vous regardez l'exemple de départ (rappelé aussi en dernière page), vous voyez que le corps a été dessiné après les pattes et que la tête a été dessinée après le corps.

On veut pouvoir :

- construire une scène a l'aide d'une `initializer_list` [2] ;
- ajouter un nouvel `Element` à une `Scene` à l'aide de l'opérateur `+=` ; p.ex. `mouton += patte_5;` ;
- fusionner deux scènes aussi avec l'opérateur `+=` ; p.ex. `troupeau += mouton_2;` (où `troupeau` et `mouton_2` sont des `Scene`) ;
- dessiner une scène sur une image, mais cela viendra en fin de question suivante.

Définissez ici la classe `Scene`, *sauf* son dessin sur une image :

Il y a plusieurs possibilités de réponses en fonction des choix effectués pour la collection hétérogène (voir le cours de la semaine 7).

La plus simple au niveau conception est d'utiliser des « pointeurs à la C » vers les éléments alloués statiquement :

2. Rappel: si nécessaire, voir l'exemple de déroulement fourni en début de donnée (et reproduit en dernière page).

```
class Scene {
private:
    vector<const Element*> elements;

public:
    Scene(initializer_list<const Element*> elements_)
    {
        for (auto f : elements_)
            if (f != nullptr) elements.push_back(f);
    }

    void operator+=(const Element* element)
    {
        if (element != nullptr) elements.push_back(element);
    }

    void operator+=(Scene const& autre)
    {
        for (auto f : autre.elements) *this += f;
    }
};
```

Cela correspond à la version « 3/3 » du cours de la semaine 7, mais ***sans*** allocation dynamique (donc **surtout pas** de `delete`!! Les éléments passés dans l'exemple de `main()` donné sont alloués *statiquement*.
(et donc *pas* besoin de redéfinir le destructeur ici))

Une autre version possible est d'utiliser la copie polymorphique (version « 1/3 » du cours de la semaine 7) :

```
class Scene {
private:
    vector<unique_ptr<Element>> elements;

public:
    Scene(initializer_list<const Element*> elements_)
    {
        for (auto f : elements_)
            if (f != nullptr) elements.push_back(f->copie());
    }

    Scene& operator+=(Element const& el)
    {
        elements.push_back(element.copie());
        return *this;
    }

    Scene& operator+=(Scene const& autre)
    {
        for (const auto& f : autre.elements) *this += *f;
        return *this;
    }
};
```

Dans cette version, il faut faire attention aux boucles : on ne peut pas copier de `unique_ptr` et donc il faut les parcourir par références constantes.

Les versions « 2/3 » du cours de la semaine 7 ne sont pas possibles car les éléments passés dans l'exemple de `main()` donné sont alloués *statiquement* et donc la gestion dynamique de la mémoire (« smart-pointers », quels qu'ils soient) **n'**est **pas** possible **sans copie**.

Autre aspect qui peut, indépendemment, être plus avancé ou non : le type de retour des `operator+()` peut être `Scene&` au lieu de `void`.

Commentaire : Plusieurs chose à dire sur cette question :

- la plus grave : plusieurs ne comprennent pas vraiment les aspects de gestion mémoire et mélangent un peu tout : typiquement confondent l'utilisation de pointeurs pour référence (« le cas d'utilisation numéro 1 » du cours) avec allocation dynamique (« le cas d'utilisation numéro 3 » du cours), par exemple libèrent de la mémoire allouée statiquement, utilisent des `unique_ptr` sans copie, etc.
- encore trop de « copié-collé » ; pensez à **réutiliser** !

  (p.ex. le `+=` d'éléments dans le `+=` de scènes)
- quelques un(e)s utilisent ici des `array` au lieu de `vector` alors que la taille n'est pas connue à la compilation ;
- aucune vérification de l'ajout de `nullptr` (dans la « version simple »)
- beaucoup ne savent pas ce qu'est une `initializer_list`.

Question 1.4 – Image [35.5 points]

Pour finir, on cherche à représenter des images : une matrice de couleurs.

On va ici distinguer l'abstraction et l'implémentation :

- au niveau abstraction, une image se comportera comme un tableau **bi-**dimensionnel de couleur : on accède à chaque pixel par deux coordonnées x et y (deux `int` pour simplifier) ;
- mais au niveau implémentation, il s'agit, pour des raisons d'efficacité mémoire, d'un unique tableau **uni**dimensionnel de couleurs.

On passe de l'un à l'autre à l'aide d'une méthode privée

```
size_t Image::access(int x, int y) const;
```

qui convertit des coordonnées de pixel dans l'index (`size_t`) permettant l'accès au choix d'implémentation. Cette méthode fonctionne pour n'importe quelles valeurs de `x` et `y`, c.-à-d. qu'elle gère elle-même les cas d'erreur et retourne toujours un `size_t` valide pour l'image.
On **ne** vous demande **pas** d'écrire la méthode `access()` (on la supposera déjà fournie).

Question 1.4.1 – Définition de base [15 points]

Pour l'initialisation d'une `Image`, on fournira sa taille horizontale (longueur L), sa taille verticale (hauteur H) et sa couleur de fond c.[3] Cette initialisation devra donc remplir le tableau (unidimensionnel) de couleurs (de taille $L \times H$) avec la couleur c.

On souhaite de plus avoir :

- accès en lecture à sa longueur et sa hauteur ;
- une méthode qui retourne la couleur du pixel (x, y) (« méthode get ») ;
- une méthode qui modifie la couleur du pixel (x, y) (« méthode set ») ;
- une méthode `dessine()` qui reçoit un `Element` et le dessine :
  elle parcourt tous les pixels de l'image, et, pour chaque pixel, s'il est dans l'élément, alors elle change la couleur du pixel pour la couleur de l'élément (et sinon il n'est pas modifié).

3. Rappel: si nécessaire, voir l'exemple de déroulement fourni en début de donnée (et reproduit en dernière page).

Définissez ici la classe `Image`, *uniquement* pour les parties indiquées ci-dessus, mais y compris les attributs :

```
class Image {
private:
    vector<Couleur> couleurs;
    int taille_x, taille_y;

    size_t access(int x, int y) const;

public:
    Image(int H, int L, Couleur fond)
    : couleurs(H * L, fond), taille_x(H), taille_y(L) {}

    void set(Couleur couleur, int x, int y)
    { couleurs[access(x, y)] = couleur; }

    Couleur couleur(int x, int y) const
    { return couleurs[access(x, y)]; }

    int largeur() const { return taille_x; };
    int hauteur() const { return taille_y; };

    void dessine(Element const& e)
    {
        for (int x = 0; x < taille_x; ++x)
            for (int y = 0; y < taille_y; ++y)
                if (e.dedans(x, y)) set(e.couleur(), x, y);
    }
};
```

Commentaire : Trois choses ici :

— Pourquoi des `vector<Couleur*>` ici : pourquoi ce pointeur ?

Chaque fois que vous mettez un pointeur (ou une référence), demandez vous : pourquoi ?
quel « cas d'utilisation » ? (cf « les trois cas d'utilisation » du cours)
est-ce nécessaire ?

Si ce pointeur est pour éviter des copies de `Couleur`, c.-à-d. pour référencer des couleurs existantes (« cas d'utilisation numéro 1 »), alors :
- ce devrait alors être `vector<const Couleur*>` ;
- ça doit être cohérent avec l'attribut de `Element` en question 1.2.1 (qui doit dont être une référence ou un const-pointeur) ;
- cela limite l'approche à n'utiliser que des couleurs connues au préalable ou créées dynamiquement avant l'utilisation de l'image (car devant être partagées ; si elles ne sont pas partagées, il n'y a **aucune** raison de mettre un pointeur).

— La méthode `access()` relève purement de l'implémentation (en auriez-vous besoin si on implémentait l'image avec un tableau à deux dimensions ?) et donc ne doit absolument pas apparaitre dans l'interface (c.-à-d. elle ne doit pas être `public`).

En plus, il était dit dans la donnée qu'elle était privée.

— Certain(e)s ont demandé s'il existait une fonction réciproque à `access()`. Ce n'est évidemment pas nécessaire. Mais surtout cela traduit la volonté de travailler sur l'implémentation au lieu de travailler sur l'*abstration* `Image`.

Question 1.4.2 – Affichage [7 points]

Ajoutez ici une définition de l'opérateur d'affichage usuel << pour les images. Il suffit simplement d'afficher la hauteur et la largeur de l'image sur une première ligne, puis d'afficher chaque couleur séparée par une espace. On fera un retour à la ligne à chaque ligne de l'image.

```
ostream& operator<<(ostream& sortie, Image const& img)
{
    const int taille_x = img.largeur();
    const int taille_y = img.hauteur();

    sortie << taille_y << " " << taille_x << endl;

    for (int y = 0; y < taille_y; y++) {
        for (int x = 0; x < taille_x; x++) {
            sortie << img.couleur(x, y) << " ";
        }
        sortie << endl;
    }

    return sortie;
}
```

Question 1.4.3 – Ajout d'une scène [13.5 points]

Pour terminer, on souhaite pouvoir facilement dessiner une scène sur l'image à l'aide d'une autre surcharge de l'opérateur `<<`. On veut par exemple pouvoir faire `dessin << mouton;` comme dans l'exemple de départ (rappelé aussi en dernière page).

On va pour cela :

1. surcharger l'opérateur `<<` pour une image et un élément (p.ex. `dessin << cercle_1;`) ; celui-ci appelle simplement la méthode `dessine()` de l'image avec l'élément comme argument ; on doit pouvoir dessiner plusieurs éléments à la suite, p.ex. :
   `dessin << cercle_1 << cercle_2;`
2. ajouter une méthode `dessine_sur()` aux `Scene` pour dessiner une scène sur une image ; cette méthode prend une image en argument (qu'elle va modifier), mais ne modifie pas la `Scene` ; elle dessine simplement chacun de ses éléments sur l'image au moyen de l'opérateur `<<` défini ci-dessus ;
3. surcharger l'opérateur `<<` pour une image et une scène (p.ex. `dessin << mouton;`) ; celui-ci appelle simplement la méthode `dessine_sur()` de la scène avec l'image comme argument ; on doit pouvoir dessiner plusieurs scènes à la suite, p.ex. :
   `dessin << mouton_1 << mouton_2;`

Définissez d'abord l'opérateur `<<` pour une image et un élément.
Ajoutez ensuite la méthode `dessine_sur()` pour la classe `Scene`.
Ajoutez enfin la définition de l'opérateur `<<` pour une image et une scène.

```
Image& operator<<(Image& img, Element const& e) {
  img.dessine(e);
  return img;
}

void Scene::dessine_sur(Image& img) const
{
    for (auto f : elements) img << *f;
}

Image& operator<<(Image& img, Scene const& s) {
  s.dessine_sur(img);
  return img;
}
```

Dans la version avec `unique_ptr`, attention à nouveau à la boucle dans `dessine_sur()`, qui doit être faite par référence constante.

Commentaire : Beaucoup trop d'erreurs ici par « réflexe sans réfléchir » : pourquoi des `ostream` ici ? Une surcharge externe est comme n'importe quelle surcharge de fonction : on peut mettre les arguments que l'on veut.

Note : certain(e)s pourraient objecter qu'il n'est pas bon de changer la sémantique des opérateurs existants, mais puisque Bjarne lui-même l'a fait (originellement `<<` est le décalage à gauche en binaire), qu'a-t-on à redire ?...