Langages Orientés Objet : C++ moderne
TP N° 2 – Les classes

Objectifs :
Ce TP vise à vous rafraîchir les notions de classes vues en S4. Un rappel succinct est à votre disposition dans le support de cours *Classes* qui est disponible sur Moodle → S5A → Langages Orientés Objet. Les transparents mêmes et la page Moodle dédiée à LOO fournissent aussi des liens vers d'autre documentation. N'hésitez pas non plus à chercher par vous même.
Ce TP contient aussi un point facultatif (écrit entre **[ ]**) ou des exercices entiers facultatifs pour les programmeurs les plus intéressés et téméraires.

En particulier ce TP traitera :
- Structures et classes.
- Encapsulation.
- Constructeurs.
- Fonctions membres et non-membres.
- Fonctions membres constantes.
- Fonctions membres et non-membres *inline*.
- Surcharge des opérateurs.
- Écriture dans un fichier.
- Lecture des données contenues dans un fichier.
- Exceptions.
- Fonctions membres spéciales :
  - constructeur par recopie,
  - constructeur par déplacement,
  - affectation par recopie,
  - affectation par déplacement,
  - destructeur.

Environnement de travail.
Pour compiler, vous devez utiliser le fichier *GNUmakefile* fourni sur Moodle dans la section « IMPORTANT – fonctionnement des Labos ». Le *GNUmakefile* permet de compiler n'importe quel ensemble de fichiers *.c* ou *.cpp* sous plusieurs plate-formes (Linux, Windows et Mac). Cela ne signifie pas que vous avez le droit d'oublier comment écrire un *makefile* !
Pour rappel, ici : https://www.enib.fr/~harrouet/dev_tools/ vous trouverez les informations nécessaires pour créer l'environnement de travail sur votre machine personnelle.

Dans un dossier de travail, nommé *s5loo*, (peut-être vous l'avez déjà créé lors du TP1), créez un sous-dossier dédié à ce TP, nommé *TP2_classes*. Placez dans ce dossier le fichier *GNUmakefile*. Ouvrez n'importe quel éditeur de texte, *kate* par exemple, et écrivez la fonction principale dans sa version avec des paramètres.
Le *main()* devra simplement afficher son nom (utilisez la variable *__func__* implicitement définie) et ses paramètres.
Sauvegardez le fichier dans le dossier *TP2_classes* en l'appelant *prog.cpp*.
Compiler votre code et assurez vous que tout fonctionne comme prévu avant de passer à l'étape suivante.

1. Structure Color
Commençons simplement. Déclarez, dans un fichier nommé *color.hpp*, une structure **Color** dans le *namespace s5loo*. Une couleur est définie par la valeur (comprise entre 0 et 255) de ses trois composantes rouge, verte et bleu. En conséquence, la structure aura trois attributs de type *unsigned char red*, *green* et *blue*.
Pour éviter à chaque fois où il sera nécessaire d'écrire en entier le type *unsigned char*, renommez ce type avec le mot *uchar* en utilisant l'instruction *using*.
Étant donné que Color est une petite structure toute simple sans invariants nous n'avons pas besoin d'appliquer l'encapsulation et d'en faire une classe. Elle sera entièrement déclarée et définie dans le fichier *color.hpp*, en conséquence toutes ses fonctions membres ou non-membres seront *inline*.

Déclarez et définissez un constructeur avec initialisation qui prend trois *uchar* pour les affecter aux attributs de la structure. Utilisez la liste d'initialisation des membres (*member initializer list*).

Testez le nouveau type Color dans le programme principale. Pour cela implémentez dans le fichier *prog.cpp* une fonction *test_color()* qui ne prend rien en paramètre et ne renvoie rien. Elle doit simplement instancier deux objets de type Color comme ceci :

```
s5loo::Color red{255,0,0};
s5loo::Color c;
```

Appelez la fonction *test_color()* depuis le *main()*. Compilez.
Vous remarquerez alors que, même si vous avez écrit parfaitement tout ce qui vous a été demandé, la compilation échue : l'instanciation de *c* cause une erreur. Cela vient du fait que nous avons fourni un constructeur avec initialisation à la structure Color et par conséquent le constructeur par défaut (qui ne prend rien en paramètre) n'existe plus. Il faudra alors le déclarer et le définir. Une couleur par défaut sera noire (0,0,0). Implémentez le constructeur par défaut en appelant par délégation le constructeur avec initialisation. Compilez et remarquez que l'erreur a disparu.

Lancez l'application. Dans l'état actuel évidemment sur la console il ne se passe rien du tout, mais cela permet de s'assurer que le code compile et s'exécute sans erreurs.

Bien que les attributs de la structure Color soient directement accessibles, pour une question de confort, nous pouvons fournir des méthodes pour les lire et le modifier. Pour les modifier, surchargez l'opérateur d'indexation (crochets []) comme ceci :

```
uchar& operator[](int i) {
  // si i est 0, 1 ou 2 renvoyer respectivement
  // la composante rouge, la composante verte ou
  // la composante bleu. Si non lancer une exception
  // de type std::out_of_range.
}
```

Elle doit être une méthode de la structure. En supposant d'avoir déclaré un objet *c* de type Color, cette méthode permet, par exemple, de modifier sa composante verte de cette façon :

```
c[1] = 75;      //ici le paramètre i de la fonction operator[] vaut 1
```

Cela est possible parce que la méthode renvoie une référence sur l'attribut désiré.
Faites la même chose (surchargez encore l'opérateur []) pour lire la valeur des attributs de la structure. Par exemple, pour lire la valeur de la composante bleu de l'objet *c*, cette méthode doit pouvoir être utilisée comme ceci :

```
uchar blue = c[2]; //ici le paramètre i de la fonction operator[] vaut 2
```

Attention ! Cette dernière méthode ne doit que lire les attributs de la structure, donc elle **doit** être une méthode constante.

Surchargez l'opérateur d'injection (<<) pour afficher à l'écran un objet de type Color. Cette fonction doit être non-membre et doit afficher les composantes de la couleur sur la même ligne, séparées par une espace blanche. Attention, l'injection d'un *uchar* requiert un cast vers *int* (par défaut il est interprété comme un *char*).
Elle nous permettra, par exemple, d'afficher l'objet Color *red* (déclaré au début de cet exercice) en écrivant :

```
std::cout << "red: " << red << '\n';
```

Évidemment, cette instruction génère une erreur si l'opérateur d'injection n'a pas été surchargé pour la structure Color. Si vous ne savez pas comment implémenter cette fonction, allez revoir les support de cours *Premiers concepts*.

Enfin, implémentez, toujours dans le ficher *color.hpp* et dans le *namespace s5loo*, une fonction non-membre *grey()* qui prend en paramètre un objet de type Color et renvoie un nouvel objet du même type dont les composantes ont été initialisées avec le niveau de gris calculé à partir de l'objet reçu : ce niveau correspond à la valeur moyenne des trois composantes rouge, verte et bleu.

Testez les nouvelles fonctionnalités en modifiant la fonction test_color() dans le fichier *prog.cpp*. Après la déclaration de *red* et *c*, affectez 165 à la composantes verte de *c*, déclarez un nouvel objet de type Color, *g*, et affectez lui la valeur retournée par l'appel à la fonction *grey()* qui reçoit l'objet *red* en paramètre.

Enfin, déclarez un dernier objet Color constant *noir*. Affichez les premiers trois objets Color et la composante rouge de l'objet *noir* à l'écran. Pour accéder à la composante rouge de l'objet *noir* utilisez l'opérateur []. Compilez et lancez l'application pour vous assurer que tout marche comme attendu.

Essayer d'appeler la fonction *grey()* sans spécifier son *namespace* d'appartenance. Vous remarquerez que la compilation et l'exécution du code marche toujours correctement. Cela est possible grâce au mécanisme qui s'appelle **ADL** (*Argument Dependent Lookup*) : le compilateur n'a pas besoin de connaître explicitement le *namespace*, il peut le déduire en regardant le type du paramètre passé. Dans les faits, indiquer le *namespace* est nécessaire principalement lors de l'instanciation d'objets ou de la manipulation d'objets déjà existants (comme *cout*, *cin*…).

Finalisez la structure proprement en définissant les 5 fonctions membres spéciales. Si vous ne vous en souvenez plus, ne vous inquiétez pas, elle sont très simples. Elles servent pour gérer la recopie, le déplacement et la destruction des objets. Dans la grande majorité des cas, il faudra simplement confirmer que leur implémentation implicite fournie par le compilateur est suffisante (en les déclarant **`=default`**), ou les interdire tout court (en les déclarant **`=delete`**). Étant donné qu'il n'y a pas de raisons d'empêcher la recopie ou le déplacement ou la destruction d'un objet de type Color, nous nous contenterons juste de confirmer l'implémentation implicite pour les 5 méthodes spéciales. Pour cela il faut ajouter à la structure les déclarations suivantes :

```
Color(const Color &) = default;            //constructeur par recopie
Color(Color &&) = default;                 //constructeur par déplacement
Color& operator=(const Color &) = default; //affectation par recopie
Color& operator=(Color &&) = default;      //affectation par déplacement
~Color() = default;                        //destructeur
```

Bien que cela semble inutile (on ne fait que confirmer ce que le compilateur fait déjà tout seul), indiquer explicitement le comportement de ces 5 méthodes spéciales est une très bonne pratique, une habitude à prendre. Cela permet de renseigner qui relie le code (collègues, patron, client, programmeur même après quelques mois…) que l'on a bien réfléchi au comportement d'un objet lors de recopies, déplacements ou destruction et que l'implémentation implicite des fonctions spéciales est déjà adaptée. Les prototypes des fonctions spéciales est toujours le même, seulement le nom du type concerné change, donc les écrire devient rapidement très facile.

2. Classe Image.

Passons aux classes. Créez un nouveau fichier d'en-tête *image.hpp* dans lequel déclarez la classe Image (sans oublier l'espace de noms *s5loo*). Cette classe doit posséder quatre attributs : une string *name_* pour le nom de l'image, deux entiers *width_* et *height_*, qui indiqueront respectivement sa largeur et sa hauteur, et *pixels_*, un vector de Color qui servira pour stocker les pixels de l'image. Chaque pixel sera représenté par un objet de type Color. N'oubliez pas d'appliquer l'encapsulation.

Commencez par déclarer et définir un premier constructeur. Ce constructeur doit recevoir en paramètre le nom de l'image, sa largeur et sa hauteur et un booléen qui indiquera si l'ensemble des pixels doit être initialisé avec des valeurs aléatoires ou pas. Ce dernier paramètre doit valoir *true* par défaut.
Ce constructeur ne peut pas être *inline*, donc créez le fichier *image.cpp* dans lequel implémentez la méthode. N'oubliez pas d'utiliser la liste d'initialisation des membres.
Il ne reste plus qu'à vérifier la valeur du paramètre booléen, s'il vaut *false* l'image ne doit contenir que des pixels noirs, si non il faut remplir le *vector* de pixels dont les composantes sont générées aléatoirement.
Vous pouvez utiliser les instructions suivantes pour créer *component* un générateur de valeurs entières pseudo-aléatoires entre 0 et 255 :

```
std::default_random_engine rndGen{std::random_device{}()};
std::uniform_int_distribution<int> component{0, 255};
```

Maintenant pour obtenir une valeur pseudo-aléatoire il faut simplement écrire *component(rndGen)*, par exemple :

```
uchar r = uchar(component(rndGen));
```

permet d'obtenir la composante rouge de la couleur.

Testez le nouveau type Image dans le programme principale. Pour cela implémentez dans le fichier *prog.cpp* une fonction *test_simple_image()* qui ne prend rien en paramètre et ne renvoie rien. Elle doit simplement

instancier deux objets de type Image comme ceci :

```
s5loo::Image img1{"random",5,3};
s5loo::Image img2{"black",3,4,false};
```

La fonction *test_simple_image()* doit maintenant être appelée par le *main()*. Modifiez le code afin que l'utilisateur puisse choisir quel test lancer (*color* ou *simple_image*) en l'indiquant dans la ligne de commande. Cela veut dire que l'application pourra être lancée comme ceci :

```
./prog color
```

ou comme cela :

```
./prog simple_image
```

Dans le premier cas, le test sur le type Color implémenté précédemment sera exécuté. Dans le deuxième cas, la fonction *test_simple_image()* sera appelée. Assurez vous aussi d'afficher un message d'erreur qui explique comment lancer correctement l'application, lorsque l'utilisateur la lance sans paramètre. Par exemple, ce message pourrait dire :

« *To run this application, specify the name of the test you want to try. For example:*
./prog color
possible tests are: color, simple_image »

Compilez et lancez l'application pour vous assurer que tout marche comme attendu.

Améliorons la classe Image. Déclarez et définissez *inline* des observateurs pour lire la valeur des attributs *name_*, *width_* et *height_*. Par contre, ne fournissez pas de méthode pour lire entièrement le vector *pixels_*, cela donnerait trop d'informations sur les choix d'implémentation que le concepteur de la classe a faits, ce qui normalement doit rester le plus invisible possible. Implémentez, plutôt, une fonction non-membre *size()* qui prend en paramètre un objet de type Image et renvoie sa dimension (servez vous des observateurs pour lire la largeur et la hauteur de l'image) et surchargez l'opérateur indexation (crochets **[]**) pour qu'il permette de lire ou écrire un objet Color situé à l'i-ème position du vecteur *pixels_*.
Assurez vous juste que l'index *i* ne dépasse pas la dimension du vecteur, si non une exception de type *std::out_of_range* doit être levée.

Recontrôlez les méthodes que vous venez d'implémenter. Doivent-elles modifier les attributs de la classe ? Si la réponse est non, n'oubliez pas de les déclarer *const*.

Surchargez maintenant l'opérateur d'injection (**<<**) pour afficher l'image à l'écran selon le format suivant :
nom
largeur hauteur
liste des pixels (un pixel par ligne)
L'opérateur d'injection doit être une fonction non-membre. Attention, son code est trop important pour qu'elle puisse être *inline*. Définissez-la dans le fichier *image.cpp*.
Pour rappel : afin d'améliorer l'encapsulation, nous essayerons toujours d'implémenter un ensemble minimale de méthodes (celles qui **doivent** manipuler directement les attributs de la classe), les autres fonctionnalités seront implémentées comme des fonctions non-membres.

Pour tester les nouvelles fonctionnalités, modifiez la fonction *test_simple_image()* dans le fichier *prog.cpp*. Affichez à l'écran les deux images instanciées. Essayez aussi d'afficher un pixel d'une des images en dehors de sa dimension, histoire de vous assurez que l'exception est bien levée, puis commentez cette instruction.

Vous avez certainement remarqué que nous n'avons doté la classe Image que d'observateurs pour lire ses attributs et qu'aucun modificateur n'a pas été implémenté. Il est fondamental de se rendre compte du fait que la classe Image doit assurer un **invariant**. En effet, le nombre des pixels doit toujours être égal à la dimension de l'image déclarée dans ses attributs *width_* et *height_*, en d'autres mots :
pixels_.size() == width_ * height_
Cela veut dire que les méthodes qui prétendent modifier la largeur ou la hauteur de l'image doivent aussi s'assurer que le vector *pixels_* contienne assez d'éléments, ce qui est tout à fait logique : si l'on veut agrandir l'image, il faut ajouter des pixels, si l'on veut la réduire, il faut en enlever. En conséquence, l'implémentation de ces modificateurs n'est pas du tout triviale. Étant juste un problème d'algorithmique, nous n'essayerons pas de le résoudre pendant ce TP, mais vous êtes libres de proposer une implémentation.
En revanche, pour l'attribut *name_*, il n'y a pas de contraintes, vous pouvez fournir une méthode qui en

permet la modification.

3. Des fonctionnalités plus complexes pour la classe Image.
Améliorons la classe Image pour qu'elle puisse charger une image à partir d'un fichier et lui appliquer un traitement simple. Déclarez un deuxième constructeur qui prend en paramètre une string *path* contenant le chemin d'un fichier d'image. Dans la liste d'initialisation des membres, initialisez le nom de l'image avec le nom du fichier que vous pouvez extraire du paramètre *path* en utilisant la fonction *imageName()* :

```
std::string imageName(std::string path) {
  std::string name;
  if(auto pos_point = path.find_last_of('.'); pos_point!=path.npos) {
    path.resize(pos_point);
  }
  if(auto pos_slash = path.find_last_of('/'); pos_slash!=path.npos) {
    name = path.substr(pos_slash+1);
  }
  return name;
}
```

Déclarez cette fonction dans le fichier *image.hpp* et définissez-la dans *image.cpp*.
Remarquez que la déclaration des deux variables *pos_point* et *pos_slash* a été faite directement dans les instructions conditionnelles (*if*), avant la condition de contrôle. Ceci est possible depuis C++17 et permet de déclarer les variables au plus prêt, uniquement là où elles sont nécessaires.
Revenons au constructeur. N'oubliez pas les autres attributs dans la liste d'initialisation des membres. Vu que vous n'avez pas encore de valeurs à leur affecter, mettez juste les accolades vides ({}). Comme expliqué en cours, cela est une bonne habitude à prendre et informe le lecteur de votre code (collègues, patron, client, vous même dans quelques temps…) que le choix de ne rien mettre dans les attributs a été fait consciemment et que ces attributs ne peuvent être initialisés que plus tard, dans le corps de la méthode.
Ensuite, le constructeur doit ouvrir en lecture le fichier, dont le chemin est donné en paramètre, et récupérer les informations nécessaires pour initialiser le reste des attributs.
Le format de fichier que nous utilisons s'appelle PPM et il est très simple, vous pouvez le voir en ouvrant dans un éditeur de texte un des fichiers *.ppm* qui vous sont fournis dans le dossier *data*. La première ligne du fichier contient (et **doit** contenir) la chaîne de caractère : P3. La seconde ligne contient la largeur et la hauteur de l'image séparées par une espace. La troisième ligne contient la valeur maximale des composantes des pixels, normalement 255 (lire cette valeur mais ne pas la prendre en compte). Les lignes suivantes contiennent les valeurs des trois composantes de chaque pixel. Vous devez lire correctement ces informations et les stocker dans les attributs *width_*, *height_* et *pixels_*.
Un exemple d'ouverture d'un fichier en lecture vous est donné en annexe.

Pour tester cette nouvelle méthode, ajoutez une fonction *test_image()* dans le fichier *prog.cpp*. Cette fonction ne renvoie rien mais prend en paramètre une string *path*. Elle déclare premièrement un objet *img* de type Image (`s5loo::Image img;`), puis vérifie la valeur du paramètre *path*. Si la string n'est pas vide, affectez à *img* un objet Image initialisé en utilisant le constructeur que vous venez d'implémenter auquel vous passez la string *path*. Si, en revanche, la string *path* est vide, affectez à *img* un objet Image initialisé avec le premier constructeur implémenté en lui passant le nom, la largeur et la hauteur de l'image que vous aurez préalablement demandés à l'utilisateur. L'image créée doit être composée de pixels de couleur aléatoire.

Modifiez le *main()* pour appeler la fonction *test_image()*. Cette fonction sera appelée en lui passant une string vide lorsque l'application sera lancée avec seulement le paramètre *image*, c'est à dire comme ceci :

```
./prog image
```

et elle sera appelée en lui passant le chemin vers un fichier *ppm* lorsque ce chemin sera renseigné à la suite du paramètre *image*, par exemple comme ceci :

```
./prog image data/planete.ppm
```

Compilez le code. Vous remarquerez alors que, même si vous avez écrit parfaitement tout ce qui vous a été demandé, la compilation échue : l'instanciation de *img* cause une erreur. Cela vient du fait que nous avons fourni des constructeurs avec initialisation à la classe Image et par conséquent le constructeur par défaut (qui ne prend rien en paramètre) n'existe plus. Pour résoudre ce problème déclarez dans la classe Image le

constructeur par défaut comme ceci :
```
Image() = default;
```
De cette manière, nous disons explicitement que nous nous contentons de l'implémentation implicite fournie par le compilateur, qui ne fait rien.

Compilez et corrigez toutes les erreurs éventuelles avant d'avancer. Lancez l'application (en lui passant les paramètres qu'il faut pour tester votre code). Évidemment dans l'état actuel ce n'est pas facile de constater ce que fait l'application vu qu'aucun affichage apparaît à l'écran. **Attention !** N'essayez pas d'afficher des images de grosse taille (comme celles fournies dans le dossier *data*), cela pourrait prendre énormément de temps !

Pour vérifier que le code fonctionne comme prévu, ajoutez plutôt une fonction *save()* qui permet de sauvegarder l'image dans un fichier *ppm*. Cette fonction, qui doit être non-membre, déclarée dans le fichier *image.hpp* et implémentée dans le fichier *image.cpp*, ne renvoie rien et prend en paramètre une référence constante sur l'objet de type Image qu'elle doit sauvegarder. Tout d'abord elle vérifie que l'objet reçu contient effectivement une image (son ensemble de pixels n'est pas vide), autrement une exception doit être levée. En suite, elle ouvre un fichier en écriture et écrit les informations sur l'image en respectant le format *ppm*. Le nom du fichier correspond au nom de l'image plus l'extension ".ppm". N'ajoutez pas de chemin, les images seront sauvegardées directement dans le dossier qui contient l'application. Cela empêchera d'écraser par erreur les images dans le dossier *data*.
Pour ouvrir un fichier en écriture vous pouvez vous baser sur l'exemple donné en annexe.

Modifier la fonction *test_image()* pour qu'elle sauvegarde l'objet *img*.

Compilez et lancez l'application pour vous assurer qu'elle marche comme prévu. Essayez d'ouvrir des fichiers *ppm* existants (dans le dossier *data*) ou de créer des images aléatoires de dimensions différentes et vérifiez que les images ont bien été sauvegardées dans d'autres fichiers *ppm*, comme prévu par la fonction *save()*.

Traitement de l'image. Ajoutez maintenant une fonction non-membre *grey()* qui crée et renvoie une image en niveaux de gris à partir d'une image reçue en paramètre. Déclarez cette fonction dans le fichier *image.hpp* et définissez-la dans le dossier *image.cpp*. Servez vous de la fonction *grey()* de la structure Color pour obtenir un pixel gris à partir d'un pixel coloré. Toujours grâce au mécanisme de surcharge, il n'y a pas d'ambiguïté entre les deux fonctions *grey()*. Pour accéder aux pixels de la nouvelle image servez-vous des surcharges de l'opérateur [] que vous avez implémentées.

Testez cette nouvelle fonctionnalité dans la fonction *test_image()*. Commencez par ajouter un paramètre booléen *greyImage* à la fonction *test_image()*. Si ce paramètre vaut *true*, appelez la fonction *grey()* en lui passant l'objet *img* créé auparavant et sauvegardez l'image qu'elle renvoie à l'aide de la fonction *save()*. Si le paramètre *greyImage* vaut *false*, aucune image grise doit être créée.
Modifiez aussi le *main()* pour que, lorsque l'utilisateur lance l'application avec le paramètre *image,* la fonction *test_image()* soit appelée avec son paramètre booléen *greyImage* valant *false.* En revanche, lorsque l'utilisateur lance l'application avec le paramètre *grey*, la fonction *test_image()* sera appelée avec son paramètre booléen *greyImage* valant *true*. Dans ce cas aussi, l'utilisateur peut spécifier le nom du fichier ppm à lire, comme ceci :
```
./prog grey data/planete.ppm
```

Compilez et lancez l'application pour vous assurer qu'elle marche comme voulu.

Pour bien finaliser la classe, il faut réfléchir aux 5 fonctions spéciales (constructeur par recopie, constructeur par déplacement, affectation par recopie, affectation par déplacement, destructeur) et indiquer explicitement quelle implémentation nous décidons de leur donner.
Réfléchissons d'abord sur la recopie. Les objets de type Image peuvent avoir une dimension assez importante et occuper des milliers d'octets. L'interdiction de la recopie a alors du sens. Chaque image n'aura qu'un exemplaire dans l'application et il ne sera pas possible de la copier, ni explicitement (dans une instruction d'affectation ou d'initialisation), ni implicitement (dans le passage par valeur de fonctions).
Pour assurer un tel comportement vis-à-vis de la recopie, nous déclarerons le constructeur par recopie et

l'affectation par recopie **=delete** :

```
Image(const Image &) = delete;             //constructeur par recopie
Image& operator=(const Image &) = delete; //affectation par recopie
```

Avant d'avancer dans notre réflexion, compilez le code et constatez que, à cause de l'interdiction de la recopie, la compilation du code maintenant échoue. En effet, lorsque dans la fonction *test_image()*, nous affectons à l'objet *img* une image chargée d'un fichier ou créée aléatoirement, le compilateur essaie de déplacer l'objet temporaire dans *img*. Mais vu que rien n'a été explicitement dit sur les fonctions spéciales de déplacement, le compilateur appelle automatiquement les fonctions de recopie (en particulier le constructeur par recopie dans cet exemple) qui ont été explicitement interdites. D'où l'erreur.
Permettre le déplacement d'un objet de type Image a tout à fait du sens, donc pour laisser le compilateur libre d'effectuer cette opération, nous devons l'indiquer explicitement en fournissant le constructeur par déplacement et l'affectation par déplacement. Leur implémentation implicite générée par le compilateur est suffisante, donc nous n'avons qu'à les déclarer **=default**, comme ceci :

```
Image(Image &&) = default;                 //constructeur par déplacement
Image& operator=(Image &&) = default;      //affectation par déplacement
```

Recompilez pour remarquer que l'erreur précédent a disparu.
Pour finir, le destructeur n'a pas besoin d'attentions particulières. Aucune ressource du système est allouée explicitement par la classe Image et le vector *pixels_* gère automatiquement la mémoire dynamique qu'il a allouée en la libérant lorsqu'il sera détruit. Donc, l'implémentation implicite du destructeur est suffisante :

```
~Image() = default;                        //destructeur
```

Bref… les exercices précédents vous ont permis de mettre en pratique plusieurs concepts. Vous avez :

- déclaré des classes dans des modules à leur dédiés,
- fourni des constructeurs pour initialiser proprement les objets,
- appliqué l'encapsulation en déclarant les attributs privés et en fournissant des observateurs et des modificateurs où c'était nécessaire (en faisant attention aux invariants),
- implémenté des fonctions membres (méthodes) et non-membres,
- implémenté des méthodes constantes et des fonctions (membres ou non) *inline*,
- surchargé plusieurs opérateurs,
- fait un choix explicite sur les 5 fonctions membres spéciales,
- ouvert des fichiers en lecture et en écriture.

4. Pour aller plus loin… (facultatif)

Prenons un problème pratique et concret et essayons de le résoudre en améliorant le code que nous avons implémenté. Les robots Nao codent les images captées par leurs caméras en format YUV. Elles sont sauvegardées dans des fichiers binaires *.yuyv* qui ne peuvent pas être visualisés directement dans un visualiseur d'images. Nous voulons lire ces images, pour les convertir en format RGB et les sauvegarder dans des fichiers PPM qui, eux, peuvent être visualisés.

Commencez par fournir une nouvelle fonction non-membre *inline RGBfromYUV()* dans le fichier *color.hpp*. Cette fonction doit renvoyer par valeur un objet de type Color initialisé avec les composantes r, g et b obtenues en convertissant les composantes y, u et v reçues en paramètre. Voici les instructions qui codent la conversion :

```
const double yd=1.1640625*(y-16), ud=u-128, vd=v-128;
const double c0=1.59765625, c1=-0.390625, c2=-0.8125, c3=2.015625;
rgb[0] = uchar(std::clamp(yd      +c0*vd, 0.0, 255.0));
rgb[1] = uchar(std::clamp(yd+c1*ud+c2*vd, 0.0, 255.0));
rgb[2] = uchar(std::clamp(yd+c3*ud      , 0.0, 255.0));
```

La fonction *std::clamp()*, est déclarée dans le fichier d'en-tête *algorithm*, donc n'oubliez pas de l'inclure.

Nous sommes prêts maintenant à modifier la classe Image pour qu'elle soit capable de charger d'un fichier binaire une image en format YUV et de la convertir en format RGB. Ajoutez un nouveau constructeur qui prend en paramètre une string *path*, dédiée à contenir le chemin conduisant à un fichier *.yuyv*. Premièrement, cette fonction doit appeler dans sa liste d'initialisation, le premier constructeur, que vous avez implémenté

pour cette classe, en lui passant le nom de l'image (extrait de la string *path* en appelant la fonction *imageName()*), la largeur et la hauteur qui sont respectivement 640 et 480 et la valeur *false* (nous ne voulons pas d'image aléatoire). Toute image générée par le robot Nao a cette taille, de plus ce n'est pas une information contenue dans le fichier *.yuyv*, comme c'était le cas pour les fichiers en format PPM. Pour toutes ces raisons, la taille de l'image est écrite en dur dans le code.
Le constructeur doit, ensuite, ouvrir en lecture le fichier dont le chemin est passé en paramètre. Vous pouvez utiliser la méthode *read()* de la classe *ifstream* pour lire d'un coup tous les octets du fichier binaire et de les stocker dans un vector :

```
const int byteCount=640*480*4/2;
std::vector<uchar> bytes(byteCount);
input.read((char*)bytes.data(), byteCount);
```

La dimension du vecteur *bytes* (640*480*4/2) dérive du fait que, dans le fichier binaire, chaque ensemble de 4 octets représente 2 pixels, donc en tout il y a 640*480*4/2 octets à lire.
Il ne reste que lire les octets dans le vecteur *bytes* par groupes de 4 pour obtenir les composantes y, u et v de deux pixels à la fois :

```
const uchar y0=bytes[i+0],
            u=bytes[i+1],
            y1=bytes[i+2],
            v=bytes[i+3];
```

(le premier pixel est composé par les valeurs y0, u et v et le deuxième par les valeurs y1, u et v).
Enfin, il faudra remplacer les pixels noirs dans le vecteur *pixels_* par ces pixels convertis en format RGB. La fonction *RGBfromYUV()* est là pour ça.

Compilez. Vous remarquerez que la compilation échoue. En effet, nous avons ajouté un constructeur qui a exactement la même liste de paramètres d'un des constructeurs déjà présents (celui qui charge une image *ppm*). La surcharge ne peut pas marcher dans ce cas, le compilateur est incapable de distinguer les deux méthodes. D'où l'erreur.

Il faut, alors, distinguer explicitement l'initialisation d'un objet à partir d'un fichier *ppm* de l'initialisation d'un objet à partir d'un fichier *yuyv*. Nous appliquons la même stratégie vue en cours à propos de la classe Point pouvant représenter des points sur le système de coordonnées cartésien et sur le système de coordonnées polaire.
Déclarez dans le fichier *image.hpp*, avant la classe Image (et toujours dans l'espace de noms *s5loo*), deux types *RGBimage* et *YUVimage* :

```
struct RGBimage { };
struct YUVimage { };
```

Ces deux structures n'ont pas besoin de contenir des champs, elles ne servent qu'à indiquer, dans les deux constructeurs qui prennent une string en paramètre, le type d'image qu'ils sont capables d'initialiser : RGB ou YUV.
Modifiez alors le premier constructeur pour qu'il reçoive en paramètre aussi un objet de type *RGBimage*, et le deuxième pour qu'il reçoive aussi un objet de type *YUVimage*. De cette façon les deux constructeurs ont une liste de paramètres bien distinctes et le compilateur sera toujours en mesure de les distinguer.
Dans l'état actuel, si vous compilez (essayez !), vous verrez qu'ils restent encore des erreurs à régler. En effet, partout où on instanciait des objets Image en lisant un fichier, le constructeur appelé ne correspond plus à aucun constructeur fourni par la classe, il ne reçoit pas assez de paramètres.
Pour corriger cette erreur, tout d'abord, il est important de comprendre que lorsqu'on modifie une classe, les changements doivent avoir le moins d'impact possible sur le reste du code. Imaginez comme cela serait embêtant si vous aviez implémenté toute une grande application en utilisant une bibliothèque et qu'au bout d'un moment, cette bibliothèque est modifiée de tel sorte qui vous oblige à corriger votre code. C'est pour éviter ce genre de situations que, bien que la structure interne d'une classe puisse être amenée à évoluer, son interface doit être modifiée le moins possible. L'interface peut augmenter (lorsque des nouvelles fonctionnalités sont ajoutées) mais pas changer.

Pour être sûrs que le code que vous avez écrit avant les modifications faites à la classe Image marche encore sans besoin de changements, donnez à votre premier constructeur (celui qui crée une image *ppm*) une valeur par défaut à son deuxième paramètre (*RGBimage tag={}*). Cela veut dire que, par défaut, lorsqu'un objet de type Image est créé à partir d'un fichier, on suppose que ce fichier est de type *ppm*. Ce qui était vrai avant et

qui nous assure que le code écrit précédemment marchera toujours sans devoir être modifié : l'interface de la classe Image n'a pas changé, elle a juste augmenté.

Recompilez et constatez la disparition de l'erreur apparue lors de la compilation précédente.

Pour tester cette nouvelle fonctionnalité, implémentez dans le fichier *prog.cpp* une fonction *test_yuv()* qui sera appelée lorsque l'utilisateur lancera l'application en passant le paramètre *yuv* suivi par le nom du fichier *yuyv* à charger, par exemple :

```
./prog yuv data/nao_capture.yuyv
```

Attention ! Si l'utilisateur ne spécifie pas le nom du fichier à ouvrir, l'application doit l'informer de l'erreur et terminer.

La fonction *test_yuv()* ne renvoie rien et prend en paramètre une string *path* contenant le chemin vers le fichier *yuyv* renseigné par l'utilisateur lors du lancement de l'application. Ensuite, elle instancie un objet *img* de type Image en passant au constructeur la string *path* et un objet de type *YUVimage*, puis elle sauvegarde *img* grâce à la fonction *save()*. Rappelez-vous que le constructeur de la classe Image et la fonction *save()* peuvent lever des exceptions !

5. Pour qui adore implémenter et en veut toujours plus… (très facultatif)

Déclarez dans le fichier *image.hpp* une fonction non-membre *sobel()* qui renvoie par valeur un objet de type Image obtenue en appliquant le filtre de **Sobel** à l'image passée en paramètre par référence constante. Cette fonction est à implémenter dans le fichier *image.cpp* (trop complexe et longue pour être *inline*).

Le filtre de Sobel permet de détecter les contours dans une image. D'un point de vue algorithmique, le principe est de calculer pour chaque pixel une valeur en fonction des pixels voisins. Plus la valeur calculée est grande et plus ce pixel est considéré comme appartenant à la frontière entre deux régions.

Dans l'algorithme il existe un filtre vertical et un filtre horizontale :

Filtre horizontale :

-1	-2	-1
0	0	0
1	2	1

Filtre vertical :

-1	0	1
-2	0	2
-1	0	1

Ainsi pour tous les pixels de l'image, on calcul une valeur tel que par exemple pour le pixel de coordonné x,y

pixel[x,y] = pixel[x-1,y-1] * -1 + pixel[x,y-1] *-2 ... + pixel[x+1,y+1] * 1;

L'idée est alors d'appliquer le premier filtre vertical, ce qui génère une nouvelle image puis appliquer le filtre horizontal ce qui génère un autre nouvelle image. Il suffit alors de faire l'addition des deux dernières images pour obtenir les contours de l'image d'origine. Pour faire la somme de deux images essayez de surcharger l'opérateur +.

Testez cette nouvelle fonctionnalité dans la fonction *test_image()* en suivant la même démarche que nous avons adoptée pour tester la fonctionnalité *grey()*. Le filtre de Sobel doit être appliqué à l'image si l'utilisateur lance l'application en lui passant le paramètre *sobel*.

Annexe lecture et écriture d'un fichier

Pour lire un fichier en C++ il faut utiliser la classe *ifstream* (Input File stream). Elle permet d'instancier

un flux (ou *stream*) vers le fichier. Puis cet objet flux peut être utilisé exactement comme l'objet *std::cin*.

```
#include <fstream>

void exampleReadFromFile(std::string path) {
  std::ifstream input(path);
  if(!input) {
    throw std::runtime_error("cannot read from file " + path);
  }
  // maintenant on peut utiliser l'objet input exactement comme l'on
  // utilise std::cin
  // par exemple si l'on veut lire un entier :
  int i;
  input >> i;
}
```

Pour écrire dans un fichier en C++ il faut utiliser la classe *ofstream* (Output File stream). Elle permet d'instancier un flux (ou *stream*) vers le fichier. Puis cet objet flux peut être utilisé exactement comme l'objet *std::cout*.

Voici un exemple sur l'ouverture d'un fichier en écriture.

```
#include <fstream>

void exampleWriteToFile(std::string name) {
  std::ofstream output(name);
  if(!output) {
    throw std::runtime_error("cannot write to file " + name);
  }
  // maintenant on peut utiliser l'objet output exactement comme l'on
  // utilise std::cout, par exemple :
  output << "P3\n";
}
```