Langages Orientés Objet : C++ moderne
TP N° 4 – L'héritage

Objectifs :
Ce TP vise à vous faire mettre en pratique la théorie présentée dans le support de cours *Héritage et Polymorphisme dynamique* qui est disponible sur Moodle → S5A → Langages Orientés Objet. Si malheureusement ce TP a lieu avant le cours, votre encadrant sera là pour vous aider. La page Moodle dédiée à LOO fournit aussi des liens vers d'autre documentation. N'hésitez pas non plus à chercher par vous même.

En particulier ce TP traitera :
- Héritage.
- Polymorphisme dynamique.
- Liaison statique et liaison dynamique.
- Utilisation d'une bibliothèque graphique.

Contexte : Ce TP est une amélioration du TP précédent. Le contexte reste donc le même, création d'une fenêtre contenant des objets graphiques. Le but final de ce travail consiste à avoir plusieurs entités qui peuvent être dotées d'une forme et de comportements différents dans la fenêtre graphique où elles seront visualisées, par exemple, elles rebondissent contre les bords, changent de vitesse si cliquées avec la souris.... Pour cela, l'approche orienté objet représente une bonne solution.
Nous commencerons, dans ce TP4, à mettre en œuvre l'héritage sur la forme des entités graphiques puis nous aborderons de façon simpliste la question des comportements qui sera reprise et complexifiée dans le TP5.
Le code implémenté pendant la dernière séance représente votre point de départ. Dans le dossier de travail, nommé s5loo, que vous avez créé lors du premier TP, créerez un sous-dossier dédié à ce TP, nommé *TP4_heritage*. Placez dans ce dossier une copie du code que vous avez développé lors du TP précédent.

1. Abstraction des classes *Circle* et *Rectangle*.
Le code que vous avez implémenté dans le TP précédent possède évidemment des répétitions. Les classes Rectangle et Circle ont du code commun, comme la position, la vitesse, la couleur, les fonctions *move()* et *draw()*. De plus Rectangle et Circle représentent des concepts similaires : des objets graphiques. Donc, il est possible de définir une classe qui est une abstraction des classes Rectangle et Circle et qui contient le code commun. On va appeler cette abstraction Shape.

Commencez à implémenter cette nouvelle classe dans un nouveau module et à y déplacer les attributs relatifs à la position, la vitesse et à la couleur des objets graphiques. Déplacez aussi les méthodes pour lire ces attributs (ce ne sont que des copier-coller) et, évidemment, effacez-les des classes Rectangle et Circle qui doivent maintenant dériver de la classe de base Shape.

Comme vu en cours, en théorie, les attributs de la classe de base Shape pourraient être déclarés *protected*, pour qu'ils soient invisibles de l'extérieur de la classe mais visibles directement par les classes dérivées. Même si cela peut sembler très confortable pour le programmeur, cette pratique rend le code d'une classe de base moins robuste : il suffit de la dériver pour accéder directement aux attributs sans passer par les méthodes fournies à ce propos et en conséquence les modifier sans respecter les précautions prises par le concepteur de la classe.

Bref, pour assurer l'implémentation d'un code robuste, c'est une bonne pratique déclarer les attributs sensibles d'une classe *private* et fournir les méthodes qu'il faut pour les lire et les modifier, même quand la classe est susceptible d'être étendue. Vous pouvez lire aussi les conseils à ce sujet donnés par les lignes guides du C++ : https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rh-protected

La fonction *move()* aussi peut être déplacée dans la classe de base, elle change *x_* et *y_* selon *sx_* et *sy_* et garde la forme dans les limites. La classe de base ne connaît pas les dimensions des objets graphiques dérivés, donc

pour l'instant nous nous contenterons de garder la position *x_*, *y_* dans la fenêtre sans nous soucier du fait que la forme peut dépasser les bords. Nous améliorerons ce code par la suite.
La classe Circle se contentera d'hériter la fonction *move()* de la classe Shape telle quelle, par contre la classe Rectangle fournira une implémentation de cette méthode en modifiant l'angle de rotation selon la vitesse angulaire et en appelant explicitement la méthode *move()* de la classe de base : *Shape::move()*.

Les deux classes dérivées se servent de la fonctionnalité *move()* fournie par la classe de base et en particulier la classe Rectangle étend ce service en ajoutant le code nécessaire pour la rotation.

Dans l'état actuel, au lancement de l'application, vous verrez à nouveau les formes graphiques apparaître et se déplacer dans la fenêtre (même si pour le moment la fonction *move()* a une implémentation plus simple de celle vue dans le TP précédent). Bien que le résultat de l'application ne soit pas différent de celui obtenu au TP 3, l'implémentation a subi une modification importante et nous avons mis en place le mécanisme d'héritage pour factoriser le code de plusieurs types vers un type commun.

2. Polymorphisme dynamique.
Le mécanisme d'héritage que nous venons d'implémenter, nous permet de traiter dès maintenant toutes les formes graphiques à travers leur abstraction. Pour nous rendre compte de cela, nous pouvons exiger que la classe Window soit composée de plusieurs Shape, sans s'intéresser au type exact de ces formes. De cette façon, si une nouvelle forme graphique, disons un triangle par exemple, venait à être ajoutée, le code de la classe Window ne subirait aucune modification. En effet, grâce au **polymorphisme dynamique**, nous savons que tout pointeur et toute référence de type classe de base peut être dirigé vers un objet de type classe dérivée, donc un seul conteneur de pointeurs vers Shape sera suffisant pour exprimer la possibilité de la classe Window de contenir plusieurs objets de type Rectangle ou Circle ou de n'importe quel autre forme graphique.
N'oubliez pas que vous devez implémenter une composition entre Window et Shape, utilisez alors un vecteur de *std::unique_ptr<Shape>* que vous appellerez *shapes_*. Remplacez aussi les méthodes *add(Rectangle)* et *add(Circle)* par une méthode *add(...)* qui permet d'ajouter des formes et modifiez la méthode *moveAll_()*, elle ne fera qu'appeler la méthode *move()* sur chaque élément du conteneur *shapes_*.

La méthode *drawAll_()* demande une attention particulière, donc pour l'instant limitez-vous à mettre en commentaire son code, on y reviendra dès que la composition entre Window et Shape sera mise en place proprement.

Modifiez la fonction *main()* dans le fichier *prog.cpp* pour qu'elle utilise la bonne méthode pour ajouter des éléments au vecteur *shapes_* de la classe Window.

Ces dernières modifications demandent un peu de réflexion dû au fait que la classe Window contient des pointeurs à Shape, donc partout où vous avez besoin de manipuler les éléments dans le conteneur *shapes_* la syntaxe doit forcement être revue et adaptée.

Assurez vous que la compilation se déroule sans erreurs ni avertissements. Évidemment, dans l'état actuel le lancement de l'application ne montrera rien du tout vu que le code de la fonction *drawAll_()* est commenté.

2.1 Méthodes virtuelles pures, classes abstraites.
Maintenant nous sommes prêts à modifier la fonction *drawAll_()*. Décommentez son code et modifiez-le pour que la fonction globale *draw()* soit appelée pour chaque élément dans le vecteur *shapes_*. Compilez votre code. Vous remarquerez toute de suite que, même si vous avez bien passé une référence sur un élément du vecteur (en tenant en compte que celui-ci contient des pointeurs), la compilation échoue. Effectivement, il n'existe pas de fonction globale *draw()* qui reçoive en paramètre une référence de type Shape, il y en a qui reçoit une référence de type Circle et une autre qui reçoit une référence de type Rectangle, mais pas de type Shape.
Une telle méthode est d'ailleurs impossible à implémenter. La classe de base Shape n'a aucune connaissance sur la forme graphique dérivée et donc nous ne pouvons pas implémenter de fonction *draw()* à son niveau. Le dessin ne peut être géré que par les classes dérivées.

Pour résoudre ce problème, la solution consiste à transformer la fonction non membre *draw()* en fonction membre de la classe Shape et de la déclarer **virtuelle pure** (ou **abstraite**). Cela veut dire que la classe Shape doit contenir la déclaration suivante :

```
virtual void draw(sf::RenderWindow& win) const = 0;
```

L'utilisation du mot clef **`virtual`** et de la syntaxe "*= 0*" sert à déclarer la méthode *draw()* virtuelle pure qui veut dire que la méthode n'a pas d'implémentation dans la classe où elle est déclarée. Une classe qui contient au moins une méthode abstraite est dite elle aussi **abstraite**, cela signifie qu'elle ne peut pas être instanciée : le compilateur n'accepte pas d'avoir des objets sur lesquels l'on pourrait appeler des méthodes privées d'implémentation, ça n'aurait simplement aucun sens.
Une classe abstraite ne peut être instanciée qu'à travers ses classes dérivées qui fournissent une implémentation pour toutes ses méthodes virtuelles pures. Il en découle que la classe Circle et la classe Rectangle doivent déclarer et définir la méthode *draw()*, il sera alors suffisant de transformer l'ancienne fonction non membre *draw()* en fonction membre. Le prototype de cette méthode doit être identique à celui de la méthode virtuelle pure déclarée dans la classe Shape, ou pour le compilateur cela ne serait rien d'autre qu'une nouvelle méthode.
Le langage C++ fournit le mot clef **`override`** pour spécifier qu'une méthode est la redéfinition d'une méthode de la classe de base, ce qui permet au compilateur de signaler une erreur si le prototype de cette méthode ne correspond à aucun prototype de méthode virtuelle dans la classe de base.
En résumant, dans la classe Rectangle et dans la classe Circle doit apparaître la déclaration suivante :

```
void draw(sf::RenderWindow& win) const override;
```

Assurez vous de lui fournir aussi une définition dans les deux classes.

Déclarer la méthode *draw()* virtuelle pure dans la classe Shape répond parfaitement à notre besoin : l'abstraction Shape nous fournit une fonctionnalité fondamentale (*draw()*) même si elle est incapable d'en spécifier une implémentation. La définition de la méthode est effectivement renvoyée aux classes dérivées.
On en déduit alors un autre avantage important de cette solution : toutes les classes dérivées s'engagent à fournir les fonctionnalités déclarées virtuelles pures dans la classe de base et toutes leurs instances peuvent être manipulées à travers un type unique, celui de la classe de base.

À retenir. En cours et pendant les TPs précédents, nous avons vu que, pour garantir le niveau d'encapsulation le plus élevé possible, il ne faut déclarer comme fonctions membres que les fonctionnalités qui ont besoin d'accéder directement aux attributs de la classe et de déclarer tout le reste comme fonction globale. Ce dernier exercice nous permet de voir que lorsqu'on est dans un cas d'héritage toute fonctionnalité d'une classe de base susceptible d'être redéfinie par les classes dérivées doit forcement être déclarée comme fonction membre virtuelle (pure ou pas).

Compilez votre code en vous assurant de corriger toute erreur et tout avertissement. Lancez ensuite l'application résultante.

2.2 Méthodes virtuelles

L'étape précédente nous a permis de revoir apparaître les formes graphiques sur la fenêtre. Vous les voyez bouger à nouveau mais vous vous rendrez compte toute de suite que les objets de type Rectangle n'effectuent plus de rotation alors qu'ils le faisaient lors de l'étape 1. Entre autre, à la destruction des Shapes le *sanitizer* nous indique que le mauvais destructeur est appelé.
Cela dérive du fait que le langage C++ applique **systématiquement** la **liaison statique** : le compilateur établit immédiatement dans le code appelant l'adresse précise et connue du code de la méthode à invoquer, donc le choix de la méthode à appeler sur un objet est fait à la compilation et dépend du type statique (indiqué à la déclaration) de l'objet même. Or, tous les cercles et les rectangles instanciés dans le *main()* sont ajoutés au vecteur *shapes_* de la classe Window. La déclaration de ce vecteur spécifie qu'il contient des *smart pointers* de type Shape (*std::unique_ptr<Shape>*), donc à la compilation le type connu des objets qui peuvent être pointés

par les pointeurs contenus dans ce vecteur est Shape. Lorsque, dans la fonction *moveAll()*, la méthode *move()* est invoquée sur chacun de ces objets, le compilateur applique la liaison statique et considère que c'est la méthode *move()* de la classe Shape qui est appelée.
Évidemment cela n'est pas ce qui est attendu. Pour que la bonne méthode soit appelée, il faut obliger le compilateur à appliquer la **liaison dynamique**, c'est à dire attendre l'exécution de l'application pour choisir la méthode à appeler, lors que le type réel des objets pointés par les éléments du vecteur *shapes_* sera connu.
Ceci est obtenu en déclarant la méthode *move()* de la classe Shape virtuelle (la déclaration de la méthode est précédée par le mot clef **`virtual`**). Toute classe dérivée qui en a besoin peut redéfinir toute méthode virtuelle de la classe de base et en informer le compilateur en indiquant explicitement le mot clef **`override`** à la fin de la déclaration de la méthode.
De la même façon, pour que le bon destructeur soit invoqué à la destruction, il faut également précéder la déclaration du destructeur par le mot clef **`virtual.`**

Modifiez le code comme expliqué et testez votre application. Maintenant les rectangles effectuent des rotations.

À retenir. La liaison dynamique est un concept fondamental de la programmation orientée objet. C'est grâce à elle que dans le code nous pouvons nous référer aux éléments à travers leur abstraction tout en étant certains qu'à l'exécution le comportement spécifique défini par le type réel des éléments sera appelé. Certains langages orientés objet (comme par exemple Java), appliquent la liaison dynamique systématiquement. Si d'un côté cela donne un certain confort au programmeur (qui ne doit plus utiliser des mots clefs pour informer le compilateur de ses intentions), de l'autre il rend l'application résultante moins performante : la liaison dynamique a évidemment un coût à l'exécution. Le langage C++ fait le choix de laisser le programmeur maître de son code et libre de décider explicitement quand appliquer la liaison dynamique (par exemple, en la limitant exclusivement aux méthodes qui sont susceptibles d'être redéfinies par les classes dérivées).

2.3 Les 5 fonctions spéciales.
En général dans un cas d'héritage la recopie et le déplacement des objets doivent être interdits. Ceci permet d'éviter le problème dit de *slicing* : ce problème apparaît lorsqu'on affecte un objet d'une classe dérivée à une instance de la classe de base, une partie de l'information est forcement perdue (*"sliced" away*).
Il suffit d'interdire la copie et le déplacement au niveau de la classe de base, automatiquement toute classe dérivée sera sujette à la même contrainte.
Nous savons déjà qu'un destructeur doit être fourni pour la règle des 5 fonctions spéciales, mais ceci n'est pas tout. Dans un cas d'héritage il faut aussi systématiquement fournir à la classe de base un destructeur ***virtuel***, même s'il n'y a rien à détruire, dans ce cas son corps sera simplement vide ou **`=default`** (ce que nous avons effectivement fait à l'étape précédente). Nous avons à présent les connaissances pour comprendre que la destruction d'un objet de type classe dérivée (atteignable par une référence ou un pointeur de type classe de base) ne causera que l'appel du destructeur de la classe de base si ceci n'a pas été déclaré virtuel (liaison statique). Or, si l'objet dérivé occupait de la mémoire dynamique que son destructeur se charge de nettoyer, à cause de liaison statique cette mémoire ne sera jamais nettoyée.
Par contre, en déclarant le destructeur de la classe de base virtuel, on informe le compilateur qu'à la destruction d'un objet dérivé (atteignable par une référence ou un pointeur de type classe de base) la liaison dynamique doit être appliquée pour le destructeur, ce qui comporte l'appel du destructeur de la classe dérivée puis celui de la classe de base. Et voilà, tout est proprement détruit !
Modifiez alors la déclaration de votre destructeur pour respecter les bonnes pratiques de programmation.

Remarquez que les 5 fonctions spéciales étant complètement définies dans la classe de base Shape, il n'est plus nécessaire de les définir à nouveau dans les classes dérivées. En effet, si la copie ou le déplacement sont maintenant explicitement interdits dans la classe de base, ils ne seront pas implicitement autorisés dans les classes dérivées.

2.4 Amélioration du code.
Sphère englobante : Notre application marche à nouveau correctement. L'héritage a été mis en place proprement ainsi que la composition entre Window et Shape. Nous pouvons alors améliorer le code en le rendant

plus propre.
Tout d'abord revenons sur la méthode *move()* de la classe Shape. Pour nous concentrer seulement sur les concepts importants, nous en avons implémenté une version plus simple, en nous assurant seulement que le point *x_*, *y_* reste dans la fenêtre, sans nous soucier du fait qu'une partie de la forme graphique peut toujours dépasser les bords de la fenêtre. Pour corriger ceci évidemment il nous faudrait avoir plus d'informations sur la forme graphique même : difficile de s'assurer que la forme reste dans la fenêtre alors qu'on ne connaît pas les dimensions. Mais la classe Shape n'a pas cette information.
La solution consiste à déclarer dans la classe Shape une méthode virtuelle pure qui renvoie le rayon de la sphère englobante de la forme graphique. Appelez cette méthode *boundingSphere()* et fournissez lui une implémentation dans les classes dérivées. Dans la classe Rectangle, elle renvoie la moitié de la longueur de la diagonale du rectangle, alors que dans la classe Circle elle renvoie le rayon du cercle.
Maintenant il est possible de reformuler la fonction *move()* de la classe Shape pour que, en prenant en considération la sphère englobante, elle s'assure que la forme graphique reste entièrement dans la fenêtre.

3. Interaction avec les formes graphiques dans la fenêtre.
Améliorons le code pour permettre un minimum d'interaction avec les cercles et les rectangles. Nous voulons pouvoir cliquer avec la touche gauche de la souris les formes graphiques et leur provoquer un changement dans le mouvement : les cercles changeront de vitesse et direction et les rectangles changeront de vitesse, direction et vitesse angulaire.

Toutes les formes graphiques doivent être sensibles au clic de la souris, pour cela nous suivrons la même démarche suivie pour le dessin et le déplacement des objets : l'abstraction Shape doit fournir une méthode, que nous appellerons *click()*, qui sera redéfinie par les classes dérivées selon leurs exigences. Ajoutez alors une telle méthode à la classe Shape, elle doit être virtuelle et recevoir en paramètre les dimensions (largeur et hauteur) de la fenêtre d'appartenance et les coordonnées du clic de la souris.
Premièrement, cette fonction doit vérifier que les coordonnées du clic de la souris soient bien sur l'objet courant (servez-vous de la sphère englobante pour cela), puis elle modifie la vitesse de l'objet en affectant à ses composantes des valeurs aléatoires (toujours comprises entre -50 et 50), la seule contrainte est que la direction de la vitesse doit être vers le centre de la fenêtre.

Conseil : pour contrôler que les coordonnées *x* et *y* de la souris sont bien dans la sphère englobante de l'objet graphique vous pouvez utiliser le code suivant :

```
const double radius=boundingSphere();
const double dx=x-x_, dy=y-y_;
if(dx*dx+dy*dy<=radius*radius) {
  //dans la sphère englobante
}
```

La classe Circle se contentera d'hériter la fonction *click()* telle quelle, alors que pour les objets de la classe Rectangle le clic de la souris doit causer aussi un changement aléatoire de la vitesse angulaire (toujours comprise entre -30 et 30).
Vous remarquez tout de suite que si la classe Rectangle redéfinit la méthode *click()* elle doit tout d'abord contrôler que les coordonnées de la souris sont bien dans l'objet courant, puis modifier la vitesse et ensuite la vitesse angulaire. Cela veut dire que le code de la méthode *click()* de la classe Shape doit être copié tel quel dans sa redéfinition de la classe dérivée, ce qui est une très mauvaise pratique.
La solution consiste à couper la fonction *click()* en deux. La première fonction (qui continuera à s'appeler *click()*) garde le contrôle des coordonnées de la souris et appelle la deuxième fonction lorsque ce contrôle est positif. La deuxième fonction (que nous appellerons *do_click()*) fournit un comportement de base en réaction au clic de la souris : changement de la vitesse avec des valeurs aléatoires (entre -50 et 50) en dirigeant toujours l'objet courant vers le centre de la fenêtre.
Toute classe dérivée qui veut un comportement spécifique en réaction du clic de la souris n'a qu'à redéfinir la méthode *do_click()*. Remarquez alors que la méthode *click()* ne doit pas être redéfinie par les classes dérivées. Pour cela, nous allons la déclarer *final*: on garde le mot clef **`virtual`** de son prototype et on y ajoute à la fin le

mot clef **`final`**. De cette façon, nous figeons son implémentation : aucune classe dérivée pourra redéfinir cette méthode. Ceci est une pratique courante et importante pour garantir le respect d'un certain comportement. Dans notre exemple, nous garantissons que le clic de la souris n'aura d'effet sur une forme graphique que lors que les coordonnées du clic sont bien sur la forme. Cette réflexion est vraie pour toute méthode de la classe qui n'a pas vocation à être redéfinie dans une classe dérivée ; une telle précaution permet de s'assurer que la sémantique des fonctionnalités de la classe de base ne soit pas modifiée par les classes dérivées. En toute rigueur, maintenant vous devriez rendre **`final`** toutes les fonctions membres qui sont non virtuelles.

Redéfinissez alors la méthode *do_click()* dans la classe Rectangle (la classe Circle se contentera toujours de son implémentation de base). La redéfinition doit d'abord invoquer la méthode *do_click()* de la classe de base (pour modifier aléatoirement la vitesse de l'objet courant, inutile de répéter un code déjà implémenté), puis modifier aléatoirement la vitesse angulaire (entre -30 et 30). N'oubliez pas le mot clef **`override`**.

Vous avez certainement remarqué que cela n'aurait pas de sens dans la classe Shape de déclarer la méthode *do_click()* publique. En effet, elle ne doit être invoquée librement depuis n'importe quelle portion de code. Pour cela, il serait tout à fait envisageable de la déclarer privée si ce n'était pour le fait que la classe dérivée Rectangle l'invoque dans sa redéfinition de la méthode *do_click()*. Pour donner la possibilité à une classe dérivée de redéfinir et en même temps d'invoquer une méthode non publique de la classe de base, la solution consiste à déclarer cette méthode protégée.