TP4 C++

CHAPELLE Victoire
SOTIR Paul-Emmanuel Binôme B3330

INTRODUCTION

Le but de ce TP est de maitriser le polymorphisme et l'héritage ainsi que de prendre conscience de ses avantages et inconvénients. Nous avons également porté une attention particulière aux move-semantics, à l'utilisation des pointeurs intelligents et à la sérialisation de Boost.

Guide de style

Notre guide de style se base sur le guide de style officiel du C++ moderne, disponible sur GitHub (https://github.com/isocpp/CppCoreGuidelines), auquel nous avons ajouté certains choix comme par exemple les commentaires au format doxygen et le guide de style de l'INSA.

Architecture du projet

Travaillant sur Visual Studio, nous avons créé un makefile générique gérant les dépendances automatiquement pour pouvoir facilement compiler le projet avec gcc. Le makefile crée des fichiers de dépendance créant à leur tour de nouvelles règles pour ce makefile.

Le projet est organisé de manière à isoler les headers, les fichiers sources et les fichiers générés :

```
/
├── src
│   └── main.cpp Rectangle.h Rectangle.cpp Segment.h Segment.cpp ...
├── makefile
├── bin ... Dossier contenant les fichiers générés (exécutable,
│           fichiers objets, fichiers de dépendances).
│   ├── debug
│   │   └── dep
│   └── release
│       └── dep
└── tests ... Dossier contenant les tests (identique au framework de
              test du TP3) .
```

SPECIFICATION DES CLASSES ET FONCTIONS PRINCIPALES

Header `Utils.h`

Le fichier '`Utils.h`' contient quelques fonctions basiques qui pourraient être communes à n'importe quels projets C++.

Nous avons notamment crée une fonction template 'mod' calculant le modulo de n'importe quel type arithmétique (dont float ou double). L'implémentation exploite le hack SFINAE (Substitution Failure Is Not An Error) pour pouvoir créer une overload pour tout type arithmétique entier signé, une autre overload pour les entiers non signé et une autre pour tout les type à virgule flottante. De plus, dans le cas des entier, la fonction est souvent plus performante que l'opérateur % simplement grâce à la vérification a >= b :

```
template <typename T, typename U,
          typename = typename std::enable_if<!std::is_arithmetic<T>::value>::type>
inline U mod(T a, T b) {
         return a % b;
}

template <typename UnsignedInteger,
         typename = typename std::enable_if<std::is_integral<UnsignedInteger>::value
                   && std::is_unsigned<UnsignedInteger>::value>::type>
constexpr UnsignedInteger mod(UnsignedInteger a, UnsignedInteger b) {
         return UNLIKELY(a >= b) ? a % b : a;
}
// ...
```

La macro '`UNLIKELY`' utilise des fonctions intrinsèques du compilateur (si disponible) pour faire en sorte que la condition soit considérée plus souvent fausse que vrai, ce qui permet d'optimiser le banchage au niveau de l'assembleur.

Classes représentant des formes géométriques

Nous avons choisi de représenter les formes géométriques (Rectangle, Segment, Polygon, Shape_union et Shape_intersection) par des types immutables et sérialisables (sérialisation détaillée dans la section suivante). De plus ces formes n'ont pas besoin d'hériter d'une même classe (grâce au mécanisme décrit dans la section 'Historique et polymorphisme') ce qui les rend d'autant plus simples à concevoir et permet de ne pas être obligé d'avoir les inconvenants du polymorphisme lors de l'utilisation de ces formes.

Puisque les formes sont immutables, lorsque l'on bouge une forme (appel à la fonction globale TP4::Move), on obtient ou nouvelle formes qui a été déplacée. Voici un exemple de prototype des fonctions Move et Is_contained (Overload pour chaque type de forme) :

```
Rectangle Move(const Rectangle& shape, coord_t dx, coord_t dy);
bool Is_contained(const Rectangle& shape, Point point);
```

De plus, les formes Rectangle, Segment et Polygon sont construites à partir des fonctions make_rectangle, make_polygon et make_segment pour que l'on ne puisse pas créer de formes invalides. Si les paramètres donnés à ces fonctions sont incorrects (polygone non convexe, bords d'un rectangles invalides ou extrémités du segment identiques) la fonction retourne 'std::experimental::nullopt'. Voici la fonction make_rectangle :

```
std::experimental::optional<Rectangle> make_rectangle(  const Point top_left_corner,
                                                        const Point bottom_right_corner)
{
         if (top_left_corner.first >= bottom_right_corner.first
                  || top_left_corner.second <= bottom_right_corner.second)
                  return std::experimental::nullopt;
         return std::experimental::optional<Rectangle>(Rectangle(std::move(top_left_corner),
                                                        std::move(bottom_right_corner)));
}
```

Le type '`std::experimental::optional<T>`' est un type du C++ 17 permettant de retourner un éventuel objet de type T sans allocations dynamiques.

Les formes Rectangle, Segment, Polygon, Shape_union et Shape_intersection ont également des surcharges de l'opérateur '<<' pour donner une description de la forme plus lisible que le résultat de la sérialisation (utilisé pour la commande LIST)

La classe Group<is_union>

Shape_union et Shape_intersection sont respectivements les types représentant une union de formes et une intersecrtion de formes. Ces classes sont les propriétaires des formes qui les composent (structure des données hierarchique) et si l'on crée une union ou une intersection de formes, alors ces formes ne seront plus desponnible du point de vue de l'utilisateur (elles ne sont plus associées à un nom unique et sont encapsulées dans l'union ou l'intersection). Il est tout à fait possible de créer une union/intersection d'intersections ou d'unions.

Les classes Shape_union et Shape_intersection sont en fait très similaires et ne sont en fait que des alias pour la classe template Group<is_union> :

```
template<bool is_union>
struct Group;

using Shape_union = Group<true>;
using Shape_intersection = Group<false>;
```

Serialisation

Pour la sérialisation (enregistrement et chargement d'un fichier), nous avons choisi d'utiliser boost::serialization qui permet de sérialiser un type facilement et de manière indépendante du type de sérialisation (fichier binaire, XML, texte, …). Ainsi, il nous suffit d'ajouter un '#define' pour pouvoir sérialiser sous forme de binaire ou textuel (nous sérialisons en XML par défaut).

Un autre avantage de la sérialisation de boost est qu'elle gère la sérialisation de pointeurs de façon à ce que la déserialisation de pointeurs vers un même objet ne crée qu'une seule fois l'objet pointé.

Pour que les formes géométriques soient sérialisables, on peut définir la fonction 'serialize' qui sera appellée par boost (de nombreuses autre manières de sérialiser existe en fonction du type de classe). Voici par exemple la fonction de sérialisation de la classe Polygon :

```
friend class boost::serialization::access;

template<typename Archive>
void serialize(Archive& ar, const unsigned int version)
{
        ar & boost::serialization::make_nvp("vertices", vertices);
}
```

'vertices' étant un vecteur des points du polygone (un 'TP4 ::Point' est un alias pour std ::pair<int, int>). On remarque que dans ce cas, boost sait déjà comment serialiser un vecteur et un std ::pair<>.

L'état de courrant de l'historique (détaillé ci-dessous) contenant toutes les formes (formes en haut de la hiérarchie des formes) est ensuite lui-même sérialisable ce qui permet d'enregistrer l'état de l'application dans un fichier facilement.

Historique et polymorphisme

L'historique (exécution des commandes undo et redo) que nous avons établi est un bon compromis entre utilisation de la mémoire et coût : L'historique conserve en mémoire les états précédents ce qui nous permet de ne pas avoir à faire un système de commandes réversibles. Cette solution peut sembler coûteuse en mémoire mais nous ne retenons en fait que les différences entre chaque états de l'historique. Le mécanisme utilisé pour implémenter cet historique est inspiré de la présentation d'un développeur de Photoshop : https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil

L'historique est indépendant des types de formes et gère en interne le polymorphisme grâce aux types privés shape_concept et shape_model<T>. Les types T de shape_model<T> seront les différents types de formes (rectangle, polygon, Shape_union, ...) encapsulés dans la classe et shape_model<T> hérite de la classe abstraite shape_concept ce qui permet le polymorphisme sur les fonctions TP4::Move, TP4::Is_contained et TP4 ::Print (Print utilise l'opérateur << des formes pour les afficher de manière polymorphique).

Les types shape_concept et shape_model<T> sont définies dans la section privée de la classe 'History_shape'. Cette classe est constructible à partir d'un objet de type T validant le concept de forme (ayant des overloads de 'Move', 'Is_contained' et de l'opérateur '<<') et crée alors à partir de cette forme un shared_pt qu'il retient. Le fait que ce pointeur est un shared_ptr est central dans le fonctionnement de l'historique : lors de copies d'un 'History_shape' ce pointeur sera copié par valeur, créant un nouveau pointeur vers la même forme géométrique. Or les formes sont immutables donc on a la garantie qu'une forme à une adresse donnée n'aura pas été modifiée et donc qu'il n'est pas nécessaire d'en faire une copie lors que l'on copie tous les 'History_shape' pour une sauvegarde de l'état de l'historique (fonction 'commit' de la classe 'Shape_history')

Une unordered_map associant un nom unique à un 'Shape_history' (pointant vers une forme) représente donc l'état actuel de l'historique. La classe 'Shape_history' n'est donc qu'un vecteur de ces états et permet d'itérer sur ces états avec les fonctions 'undo' et 'redo' ainsi que copier l'état actuel (passer à l'état suivant avec 'commit'). Lors de cette copie les formes ne seront pas réellement copiées (seulement les pointeurs). Par contre des copies de formes seront réalisées lors de leur modification (appel à 'Move' par exemple). On obtient donc un historique ne retenant que les changements, n'ayant pas de code spécifique aux commandes (pas de code pour inverser une commande par exemple) et non intrusif pour les l'implémentation des formes (Rectangle, Polygone, …) car le polymorphisme nécessaire est mis en place en interne ce qui rend l'ajout de nouvelles commandes ou de nouvelles formes très facile.

Classe Scene

Cette classe est utilisé par le main qui lie l'entée standard pour exécuter chaque commande. La classe Scene contient donc l'historique et manipule ou sérialise les formes tout en appelant la méthode commit de l'historique lorsque l'on va exécuter une action annulable.

CONCLUSION ET AMELIORATIONS POSSIBLES

Nous avons appris à créer du code réutilisable (notamment en séparant l'utilisation polymorphique des types de ces types) et nous avons mieux compris la sérialisation, l'intérêt des types immutables et ce que pourraient apporter les concepts du C++ 17 (permettant de définir des contraintes sur des paramètres templates plus facilement que par SFINAE ou la méta-programmation notamment).