Xu Frédéric

Rapport du projet c++

Sommaire :

Table des matières

TASK 0 :

A. Executions

Les différentes commandes de notre programme après son exécution sont :
La touche ‘c’ pour ajouter un avion
La touche ‘x’ ou ‘q’ pour quitter le programme
La touche ‘f’ permet de mettre la fenêtre en plein écran

- L'avion apparaît, il atteri sur la piste d’atterrissage, se fait entretenir dans un terminal puis dès que c'est fini il quitte le terminal pour redécoller.
Dans la console on nous annonce chaque étape, c'est-à-dire le nom de l'avion qui atteri (‘avion’ is now landing ...), le début de sa maintenance (now servicing ‘avion’), la fin (done servicing ‘avion’) et le redécollage (‘avion’ lift off).

-Nous n'avons pas plus de 3 places de maintenances dans l'aéroport donc uniquement 3 avions à la fois peut atterir, les autres continuent à voler autour de l'aéroport.

B. Analyse de code

La classe Aircraft représente un avion et lorsqu'on crée un objet aircraft, elle permet de créer un avion

const std::string& get_flight_num() const : Obtient le numéro de vol de l'avion
float distance_to(const Point3D& p) const : Donne la distance en float de la position de l'avion au paramètre p.
void display() const override : Permet d'afficher graphiquement l'avion dans la fenêtre de la simulation
void move() override : Permet de déplacer l'avion

La classe Aircraft_type représente un type d'avion, son rôle est de différencier les avions.
La classe Airport représente l'aéroport, son rôle est d'accueillir des avions depuis le piste d'atterrissage.
Tower& get_tower() : Renvoie la tour de contrôle lié à l'aéroport.
void display() const override : Permet d'afficher l'aéroport
void move() override : Permet de lancer la méthode move() de chaque terminal

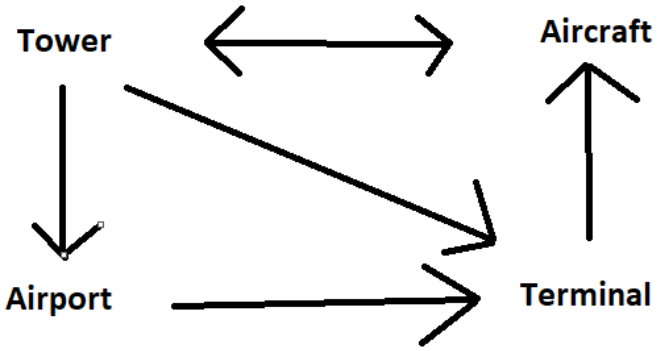
La classe Airport_type représente le type d'aéroport, son rôle est de différencier les aéroport.
La classe Point2D représente un point en 2 dimension, son rôle est faire toutes les calculs de la fenêtre
La classe Point3D représente un point en 3 dimension.
La classe Runway représente la piste d'atterrissage, son rôle est d'accueillir des avions.
La classe Terminal représente le lieu d'entretien des avions, il regroupe notamment les fonctions liés à celle-ci comme le fait qu’un terminal est
déjà utilisé, l’état du débarquement, d’assigner un terminal à un avion, de commencer ou de terminer le débarquement.
bool in_use() const : Renvoie si oui ou non le terminal est déjà en service avec un avion
bool is_servicing() const : Renvoie si oui ou non le terminal à fini son entretien avec l'avion

void assign_craft(const Aircraft& aircraft) : Permet d'assigner l'avion en paramètre dans la variable de la classe
void start_service(const Aircraft& aircraft) : Permet de lancer l'entretien de l'avion en paramètre et nous le signal
void finish_service() : Permet de signaler que l'avion a fini son entretien
void move() override : Fait progresser le temps de l'entretien

La classe Tower permet de créer une tour qui se charge de l'entretien de l'avion
WaypointQueue get_instructions(Aircraft& aircraft) : Vérifie que l'avion est dans un terminal, sinon le conduit en lui donnant le chemin jusqu'au terminal qu'on a réservé pour lui.
void arrived_at_terminal(const Aircraft& aircraft) : Nous signal qu'un avion est arrivé dans un terminal et commence l'entretien.

La classe tower_sim permet d'initialiser le simulateur en initialisant les tours et les avions.
La classe waypoint permet d'indiquer si l'avion est dans les airs ou sur terre et aussi la position de l'avion

UML des relations :



Le conteneur de la librairie standard qui a été choisi pour représenter le chemin est std::deque<Waypoint> pour le conteneur, on a choisi deque car pour une queue, on aurait pu utiliser std::queue, mais on a choisi std::deque parce qu'on veut rajouter des Waypoints à la fin et au debut de la queue.

- Les classes Aircraft et Tower sont impliquées dans la génération du chemin d'un avion par les méthodes get_instructions et move.
On utilise une la librairie <deque> pour ajouter des éléments assez rapidement et qu'on utilise deque pour pouvoir ajouter des éléments au début et à la fin de la queue

C. Bidouillons

1. Ils sont définis dans la classe aircraft_types

2. La variable qui contrôle le framerate est ticks_per_sec se situant dans le namespace GL. Lorsqu'on essaye de mettre le framerate à 0, le programme rencontre un problème, notamment à cause de la division par
0, on décide donc de faire un boolean avec un if et else s'il est vrai et modifier directement glutTimerFunc pour mettre en pause le programme.

3. C'est SERVICE_CYCLES dans config.hpp

4. On utilisera un boolean dans la classe Aircraft qui sera a faux par défaut et passera a vrai avant de redécoller. Par la suite on vérifie dans timer de la classe opengl_interface si ce boolean est vrai ou faux. Si elle est vrai, on le détruit dans move_queue. Pour que tout fonctionne, il nous faut le virtual boolean se trouvant dans dynamic_object.

5. DynamicObject est un set donc ce n'est pas nécessaire tandis que Displayable en le mettant static inline, il sera accessible et pourra être modifier où l'on veut.

6. On utilisera une map ‘std::map<const Aircraft*, size_t>’ avec l’avion en clef et ainsi retrouver le terminal facilement.

D. Théorie

1. On utilise une map privé dans la classe tower pour qu'il s'occupe directement d'associer des avions et des terminaux, elle est private donc personne d'autre pourra l'utiliser.

2. Pour éviter tout problème lié au référence, on veut notamment pas modifier sa valeur.

Task 1:

Analyse de la gestion d'avion:

Il n'y a pas moyen, sauf si on cherche dans la map de AircraftToTerminal puis dans chaque aircraft les numero de vol

Objectif 1 - Référencement des avions

A - Choisir l'architecture

Faire un AircraftManager est plus propre et plus lisible, cela permet d’éviter de surcharger une classe déjà existante, plus simple à manipuler et pouvoir déléguer une tâche précise pour la classe contrairement à utiliser une classe déjà existante.
Le seul désavantage est que cela créé plus de classe et donc faut réussir à les maintenir mais aussi de créer plus d’instance.

B – Déterminer le propriétaire de chaque avion

1. ‘void timer(const int step)’ de la classe ‘opengl_interface.cpp’ est le responsible de la destruction des avions. Il appelle un destructeur de aircraft et détruit donc le dynamic objet.

2. Nous avons ‘display_queue’, ‘move_queue’ , ‘reserved_terminals’ (Tower) et ‘const Aircraft* current_aircraft’ qui contiennent une référence sur un avion,

3. En ajoutant des destructeurs d’avions dans les classes qui stock les avions, ainsi on supprimera l’avion de la liste mais aussi en appellant un destructeur d’avion il sera aussi effacé de la mémoire. L’owner de chaque avion est ‘move_queue’.

4. Pour le faire, il faudrait qu’aircraft ait accès à son aircraftManager mais cela n’aurait pas de sens donc c’est bien aircraftManager qui s’en occupera.

C – C’est parti !

On stockera les avions dans un vecteur de unique_ptr d’aircrafts. On n’a plus besoin de les ajouter dans la move_queue car nous avons maintenant un vecteur d’avion. On utilisera notre propre fonction move qui va utiliser tous les avions de notre vecteur. Ainsi on pourra remettre la fonction timer a son état d’origine et le mettre en virtual car on veut utiliser celle d’aircraftManager. Cela est possible parce qu’aircraftManager hérite de DynamicObject.

Objectif 2 – Usine à avions

A – Création d’une factory

La classe aircraftFactory reprendra les fonctions create_aircraft, create_random_aircraft, airlines et aircraft_types déjà existante car c’est la factory qui s’occupera de la création des avions.
Mais cela n’est pas suffisant pour faire fonctionner le programme, nous devons créer un struct de ContextInitializer dans la classe TowerSimulation. Nous allons créer un attribut ContextInitializer dans TowerSimulation en privé qui sera au-dessus des attributs aircraftManager et aircraftFactory puis le définir en le mettant en référence dans les paramètres du constructeur de TowerSimulation. Ainsi il sera appelé avant la factory. On supprime bien entendu les inlines car nous ne voulons plus de variable global.

B – Conflits

Pour régler ce conflit de nom d’avion, il suffit d’utiliser un set pour les noms d’avions. Ainsi il n’est pas possible d’avoir de doublon de nom. Nous pouvons donc à chaque création d’avion vérifier que le nom d’avion ne se trouve pas déjà dans le set de nom pour ensuite l’ajouter dans le set de nom d’avion et finir sa création.

Task 2 : Algorithmes

Objectif 1 – Refactorisation de l'existant

A – Structured Bindings

On va tout simplement utiliser [first, second] à la place de ks_pair pour la structure binding.

B – Algorithmes divers

1. On créé tout simplement un prédicat qui bouge l'avion et renvoit un booléan pour savoir si il doit être détruit ou non pour ensuite le stocker dans une variable avec 'remove_if'. Ainsi, on pourra effacer facilement les avions lorsqu'ils ont leur 'delete_aircraft' a true avec erase.

2. Pour cela, on ajoutera une nouvelle méthode dans aircraftManager, 'airline_counting' qui utilisera 'count_if' provenant de <algorithm> et retournera le nombre d'avion appartenant à l'airlines[x], ce x se récupère grâce au 'get_airlines' de aircraftFactory qui retourne tout simplement une string_view de l'airlines[x].

C – Relooking de Point3D

1. Dans <algorithm> nous avons une fonction 'std::transform' permettant de modifier chaque valeur de l'array, ainsi en partant du début à la fin on va multiplier chaque valeur par le scalar.

2. On utilisera une méthode similaire sauf qu'à la place du lambda, on utilisera des fonctions tel que std::minus<float> {} pour l'operator-= et std::plus<float> {} pour l'operator+=, l'idée reste la même, on modifie chaque valeur de l'array par rapport à l'opérateur voulu.

3. Pour length, on utilisera la fonction inner_product dans sqrt, en partant de la valeur initial : float 0. , il retournera le résultat de l'accumulation de la valeur initial et le produit de tous les pairs d'elements en partant du début jusqu'à la fin de l'array values.

Objectif 2 – Rupture de kérosène.

A – Consommation d'essence

On créé un attribut fuel qui sera un unsigned int (ainsi pas besoin de vérifier qu'il soit positif) représentant le carburant d'un avion. On utilisera std::rand() pour qu'à l'initialisation d'un avion, il possède un niveau de carburant aléatoire entre 150 et 3000. Puis dans move on décrémentera fuel à chaque appel à move et lorsque fuel atteindra 0, on mettra un boolean représentant sa destruction à true (ici delete_).

B – Un terminal s’il vous plaît

1. has_terminal retournera simplement un boolean qui nous dira si un terminal a déjà réservé pour l’avion appellant la méthode.

2. is_circling retourne un boolean qui est tout simplement l’inverse de has_terminal, s’il n’a pas de terminal cela veut dire qu’il tourne toujours autour de l’aéroport.

3. Dans la méthode reserve_terminal dans la classe tower, on appelle reserve_terminal de la classe airport et on vérifie qu’il est possible de renvoyer un chemin vers un terminal et on le renvoit, si cela n’est pas possible on renvoit un chemin vide.

4. On va ajouter dans la méthode move() d’aircraft une condition vérifiant si is_circling est true et is_service_done est a false (Donc que l’avion est en attente) avec au moins 100 de fuel, ainsi on l’ajoutera dans la queue avec un std::move() car c’est un unique_ptr.

C – Minimiser les crash

Dans aircraftManager, la méthode move commencera par un std::sort qui permettra de trier dans l’ordre croissant de fuel par rapport a s’ils ont réservé ou non un terminal. C’est-à-dire qu’on aura d’abord tous les avions qui ont été réservé dans le début de la liste dans l’ordre croissant de fuel puis ceux qui ont n’ont pas été réservé dans l’ordre croissant de fuel.

D – Réapprovisionnement

1. Je fais tout simplement un booléan qui return si le niveau de fuel est en dessous de 200.

2. Pour la fonction ‘get_required_fuel’, j’utilise std::accumulate de <numeric>, il parcourera toutes les valeurs de aircrafts, on met sa valeur initial à 0 puis on utilise un lambda pour si l’avion est au sol et n’a plus beaucoup de fuel, on ajoute à la valeur initial le nombre de fuel manquant pour être au maximum sinon on ne fait rien.

3. Pour pouvoir avoir accès à l’aircraftManager dans la classe airport, on l’a tout simplement rajouté dans les attributs en privé.

4. En suivant la consigne, nous rajoutons de l’essence dont il a besoin pour être au maximum dans le fuel de l’aircraft en réduisant le fuel_stock possible avec évidemment une condition qui nous indique quel avion a été réapprovisionné avec sa quantité d’essence utilisé. fuel_stock est un unsigned int pour éviter qu’il soit négatif.

5. On fait en sorte que si l’aircraft est bas en essence et qu’il est au sol (donc dans un terminal), on appelle refill sur l’avion en question.

6. Si next_refill_time vaut 0, on ajoute la valeur d’ordered_fuel dans fuel_stock, puis on vérifie le plus petit entre get_required_fuel et 5000 pour modifier la valeur de ordered_fuel par celle-ci puis on reinitialise la valeur de next_refill_time par 100. Et si next_refill_time ne vaut pas 100, on le décrémente. On a notamment modifié le move de terminal pour qu’il prenne un unsigned int pour fuel_stock en paramètre et ainsi utiliser refill_aircraft_if_needed directement dans le move.

Task 3: Assertions et exceptions

Objectif 1 – Crash des avions

1+3. Pour cela, on utilise un try-catch pour attraper l'erreur et éviter qu'elle se propage jusqu'à la fin du programme. On l'utilise sur le move d'aircraftManager et on catch l'AircraftCrash puis on affiche l'erreur avec un crash.what() dans un std::cerr.

2. On créé un attribut dans aircraftManager correspondant au compteur de crash qui sera à 0 par défaut, puis on l'incrémente dans le try-catch. Pour pouvoir le récupérer dans la classe tower_sim, nous avons créé un getter pour cela.

Objectif 2 – Détecter les erreurs de programmation

Au trie d'aircrafts je vérifie que chaque aircraft n'est pas null.
A l'ajout d'un aircraft j'ai un assert pour vérifier qu'il ne soit pas null.
Airling_counting vérifie que son paramètre n'est pas vide.
A l'ajout d'un flight_number, je vérifie qu'il n'est pas null.
Je vérifie que get_airlines soit sur la bonne range.
A la création d'aircraft dans l'aircraftFactory, je vérifie que l'aéroport lié est initialisé.

Task 4 :

Objectif 1 - Devant ou derrière ?

2. On créé un template sur la fonction add_waypoint avec un boolean constant en paramètre et on appelle add_waypoint avec add_waypoint<false> car il était déjà false auparavant.

Objectif 2 – Points génériques

1. On va faire une nouvelle struct avec un template qui prend une taille et un type pour les points demandés, on utilisera template<const int size, typename t> avec t qui sera ici le type voulu (float par exemple).

3. J'utilise l'alias type pour représenter un Point avec sa taille et le type mis. On remet tous les fonctions utilisés par Point2D et Point3D par le type que j'ai créé. Après avoir fini avec la classe template on utilise un alias pour Point2D et un alias pour Point3D utilisant la classe Point avec leur nombre d'argument (donc respectivement 2 et 3) et float

4. Les erreurs se produisent seulement maintenant car on vient de les implémenter et l'erreur provient surement d'un problème de type .

5. On va mettre beaucoup de static_assert ici, dans les 2 constructeurs et pour x(), y() et z(). Les static_assert dans les 3 fonctions sont pour vérifier si la taille de donnée au template sont correctes et éviter des problèmes. Et les static_assert pour les 2 constructeurs sont pour vérifier que ce sont bien les types demandés, c'est-à-dire que c'est bien Point2D qui utilise le constructeur avec 2 types et Point3D pour celui avoir 3.

6. Pour utiliser le variadic template ici, je créé un nouveau template <typename… Args> puis un Point avec comme premier argument le type et en second une référence universelle avec && d'une array values du même type que le premier argument de Point et un std::forward<Args>. En faisant ceci et pour que cela marche j'ai du modifié tous les Point en un float car dans notre programme on utilise Point2D et Point3D comme des int ou des float selon la situation. Mais ici avec un variadic template on ne peut pas procéder ainsi, on aurait des erreurs de types. Donc j'ai du modifié dans toutes les classes utilisant Point2D et Point3D (texture, aircraftFactory, airport_type, geometry, runway, tower_sim) les valeurs int en float. En faisant cela on peut supprimer nos 2 constructeurs de Point pour utiliser seulement le variadic template.

Les questions supplémentaires :

Pour les choix que j'ai fait sur ce projet, ils sont répondu dans la partie d'avant celle où je réponds aux questions.

Les situations où j'ai bloqué sont au tout début à la suppression de l'avion mais la réponse se trouve dans les réponses données dans le task 0. Pour l'objectif 2 sur la kérosène je pensais avoir réussi au début mais je me suis rendu compte vers la fin en faisant mon rapport que si je laissais tourner le programme assez longtemps je rencontrais des erreurs qui n'avaient rien à voir avec mon code, ca m'a pris beaucoup de temps pour résoudre le problème, j'ai même du prendre une autre approche pour cette partie, elle est expliqué dans les réponses aux questions. Pour le variadic template j'avais un énorme doute sur la manière de procéder mais aussi je me demandais si c'était correct de modifier tout le code en changeant tous les int en float pour que ca marche. Je ne sais toujours pas si cela était la bonne solution. Sinon généralement c'est des blocages bêtes, par exemple oublié de changer une variable a true, ca m'est arrivé pour is_done_servicing, à cause de cela mon avion décollait puis réatterrissait instantannément en boucle sans que je comprennes la raison car il n'y avait pas réellement d'erreur de code.

Par rapport à ce projet, j'ai beaucoup aimé cette nouvelle approche, partir de quelque chose et de l'améliorer, comprendre le code d'une autre personne qui fonctionnait dans une nouvelle langue et de l'améliorer. Avoir une ligne directrice de chaque chose à faire pour ne pas se perdre.

Le fait que chaque task ne sont pas équivalent en terme de durée / difficulté par exemple le task 2 et beaucoup plus complexe et long que le task 3 était un peu démotivant lorsqu'on faisait le task 2. On ne s'attendait pas à que le task 3 soit aussi court. Sinon essayé de comprendre un bug qui apparaît sans comprendre la raison car il y a pleins de choses que je n'ai pas codé est très frustrant, je ne sais jamais si je suis autorisé à modifier tel fonction ou si en faisant cela je me mets dans d'énorme

difficulté pour la suite ou qu’une modification fasse des répercussions non voulu et qu’on ne le voit que par la suite.

J’ai appris comment gérer un projet en cpp, utiliser les différents outils appris en cours dans un cas concret.