INSTITUT
D'ÉLECTRONIQUE
ET D'INFORMATIQUE
GASPARD-MONGE

tower_control

Master 1 Informatique
Étudiant : Decilap Jason
Professeur : Noël Céline & Marsault Victor

Programmation générique en C++
Groupe1
2021-2022

Table des matières

Réponses

Task0 : Se familiariser avec l'existant

A – Exécution

Allez dans le fichier `tower_sim.cpp` et recherchez la fonction responsable de gérer les inputs du programme.

Sur quelle touche faut-il appuyer pour ajouter un avion ?

Comment faire pour quitter le programme ?

A quoi sert la touche 'F' ?

La fonction TowerSimulation::create_keystrokes est responsable de gérer les inputs du programme.

La touche 'C' permet d'ajouter un avion.

La touche 'X' ou 'Q' permet de quitter le programme.

La touche 'F' permet de mettre le plein écran.

Lorsqu'il y a un avion son comportement est le suivant :

1. L'avion vole
2. L'avion atterrie
3. L'avion se fait entretenir
4. L'avion redécolle
   (On recommence le cycle)

Ajoutez un avion à la simulation et attendez.

Quel est le comportement de l'avion ?

Quelles informations s'affichent dans la console ?

Pour l'avion BA9850 les informations suivantes s'affichent dans la console :

1. BA9850 is now landing... (Atterrissage)
2. now servicing BA9850... (Début de l'entretient)
3. done servicing BA9850 (Fin de l'entretient)
4. BA9850 lift off (Décollage)
   (On recommence le cycle)

Ajoutez maintenant quatre avions d'un coup dans la simulation.

Que fait chacun des avions ?

Lorsqu'il y a 4 avions :

- Il y a 3 avions maximum qui peuvent atterrir et aller à l'entretient.
- Les avions supplémentaires attendent en continuant de voler puis atterrissent quand il y a une place de libre pour l'entretient.

B - Analyse du code

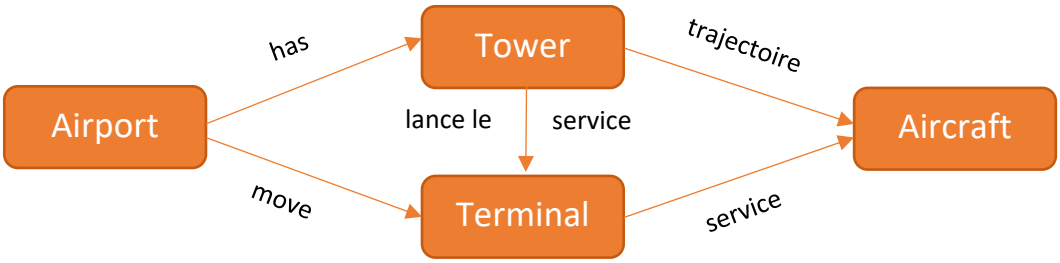
Listez les classes du programme à la racine du dossier src/.

Pour chacune d'entre elle, expliquez ce qu'elle représente et son rôle dans le programme.

Class	Représentation	Rôle
Aircraft	Avion	Créer des avions avec ses différentes propriétés.
AirportType	Type d'aéroport	Avoir différent type d'aéroport.
Airport	Aéroport	Créer un aéroport avec ses éléments
Terminal	Terminal de l'aéroport	Gérer les fonctionnalités apporter par un terminal sur les avions.
TowerSimulation	Action de l'utilisateur	Déclencher les fonctionnalités en fonction de la touche appuyée.
Tower	Tour de contrôle	Contrôler les fonctions des terminaux et des avions
Waypoint	Point de la trajectoire d'un avion	Point par lequel un avion va passer, une suite de ces points sera une trajectoire.

Pour les classes Tower, Aircaft, Airport et Terminal, listez leurs fonctions-membres publiques et expliquez précisément à quoi elles servent. Réalisez ensuite un schéma présentant comment ces différentes classes interagissent ensemble.

Class	Fonction	Utilité
Tower	Tower	Constructeur qui affecte un Airport à la Tower.
	get_instructions	Prend en paramètres une référence Aircraft et renvoie une deque de la trajectoire que l'Aircraft devra prendre : Se diriger vers l'aéroport si l'Aircraft est loin Faire un tour en attendant qu'un Terminal se libère Sortir du Terminal pour s'envoler.
	arrived_at_terminal	Prend en paramètre une référence constante Aircraft et ne renvoie rien, lance le service du Terminal sur lequel se situe l'Aircraft.
Aircraft	Aircraft	Constructeur qui affecte un type, identifiant, position, vitesse, Tower à l'Aircraft.
	get_flight_num	Renvoi une référence constante de l'identifiant du Aircraft.
	distance_to	Prend en paramètre une référence constante d'un Point3D et renvoie la distance entre l'avion et ce point.
	display	Affichage de l'Aircraft.
	move	Fait déplacer l'avion vers son prochain waypoints, s'il n'y en a aucun alors on demande a la Tower la prochaine destination.
Airport	Airport	Constructeur qui affecte un type, une position, Terminal etc…
	get_tower	Renvoie une référence Tower qui représente la tour de contrôle.
	display	Affichage de l'Airport.
	move	Fait appel à la méthode move de chaque Terminal
Terminal	Terminal	Constructeur qui affecte une position a un Terminal
	in_use	Renvoie un boolean qui dit si le Terminal est libre.
	is_servicing	Renvoie un boolean qui dit si le Termial est en train de travailler.
	assign_craft	Prend en paramètre une référence constante Aircraft et lui assigne la référence à son current_aircraft.
	start_service	Prend en paramètre une référence constante Aircraft, cette fonction annonce que le Terminal commence le service.
	finish_service	Annonce la fin du service et libère son current_aircraft.
	move	Fait avancer le processus de service de l'avion, incrémente son service_progress.



Quelles classes et fonctions sont impliquées dans la génération du chemin d'un avion ? Quel conteneur de la librairie standard a été choisi pour représenter le chemin ? Expliquez les intérêts de ce choix.

Lors de l'appel de Aircraft::move si la deque est vide on fait appel à Tower::get_instruction.
Tower::get_instruction :

- Appelle Tower::get_circle() qui renvoie une trajectoire circulaire pour faire patienter l'Aircraft si il n'y a pas de Terminal disponible.
- Appelle Airport::reserve_terminal pour avoir la trajectoire vers le Terminal réserver pour l'Aircraft
- Appel Airport::start_path pour faire décoller l'Aircraft de l'Airport.

Le conteneur std::deque a été choisit pour représenter un chemin car il permet :

- L'ajout d'éléments à la fin du conteneur.
- Réception d'éléments au début du conteneur.
- Complexité O(1).

C - Bidouillons !

1. **Déterminez à quel endroit du code sont définis les vitesses maximales et l'accélération de chaque avion. Le Concorde est censé pouvoir voler plus vite que les autres avions. Modifiez le programme pour tenir compte de cela.**

Dans le fichier aircraft_types.hpp dans la struct AircraftType nous avons les champs :

- float max_air_speed qui représente la vitesse maximale de l'Aircraft
- float max_accel qui représente l'accélération maximale de l'Aircraft

2. **Identifiez quelle variable contrôle le framerate de la simulation. Ajoutez deux nouvelles entrées au programme permettant d'augmenter ou de diminuer cette valeur. Essayez de maintenant mettre en pause le programme en manipulant ce framerate. Que se passe-t-il ?**

   Ajoutez une nouvelle fonctionnalité au programme pour mettre le programme en pause, et qui ne passe pas par le framerate.

Dans le fichier config.hpp la variable constexpr unsigned int DEFAULT_TICKS_PER_SEC permet de contrôler le framerate de la simulation.

La touche 'U' permet d'augmenter le framerate.
La touche 'D' permet de diminuer le framerate

Dans la méthode GL::timer nous avons une division par ticks_per_sec, donc si nous dé-incrémentons trop nous aurons une division par 0 et cela provoquera des évènement inattendue. De plus il faut faire attention au dépassement de capacité.

La touche 'P' permet de mettre le programme en pause et de reprendre.

3. **Identifiez quelle variable contrôle le temps de débarquement des avions et doublez-le.**

C'est la variable constexpr unsigned int SERVICE_CYCLES dans le fichier config.hpp

4. **Lorsqu'un avion a décollé, il réattérit peu de temps après. Faites en sorte qu'à la place, il soit retiré du programme.**

   Indices :

   A quel endroit pouvez-vous savoir que l'avion doit être supprimé ?

   Pourquoi n'est-il pas sûr de procéder au retrait de l'avion dans cette fonction ? A quel endroit de la pile d'appels pourriez-vous le faire à la place ?

   Que devez-vous modifier pour transmettre l'information de la première à la seconde fonction ?

On peut savoir que l'avion doit être supprimer dans la méthode Aircraft ::operate_landing_gear.
Il n'est pas sûre de le supprimer ici car il pourrait y avoir bug/crash car une autre class essaierai d'y avoir accès. On pourrait le faire dans la Class GL::timer.
Pour transmettre l'information de la 1ère a la 2nd fonction on renverra un booleen.

On change le nom de la méthode GL::DynamicObject::move en GL::DynamicObject::update qui renvoie true si on a bien mis l'objet a jour et false si on doit supprimer l'objet.

Pour cela on ajoute un champs bool service_done qui indique si on a fini l'entretient de l'Aircraft dans un terminal de la base.
Par défaut il est à false et on le met à true dans Tower::get_instructions dans la condition qui indique que l'entretient de l'avion est terminé.
Dans le méthode GL::timer on parcourt la liste des DynamicObject puis au fur et à mesure des updates on supprime ou non les Aircraft dont la méthode Aircraft::update renvoie true.

5. **Lorsqu'un objet de type Displayable est créé, il faut ajouter celui-ci manuellement dans la liste des objets à afficher. Il faut également penser à le supprimer de cette liste avant de le détruire. Faites en sorte que l'ajout et la suppression de display_queue soit "automatiquement géré" lorsqu'un Displayableest créé ou détruit. Pourquoi n'est-il pas spécialement pertinent d'en faire de même pour DynamicObject?**

Dans le fichier displayable.hpp on met la variable display_queue dans la Class Displayable en la mettant static pour y avoir accès dans les autres fichiers. On ajoute un displayable dans la display_queue dans le constructeur et on la retire dans le destructeur. On n'oubliera pas de supprimer le code ancien qui ajoutaient et supprimaient les objects de la display_queue.

Il n'est pas aussi pertinent de le faire dans la Class DynamicObject par soucis de lisibilité car on a des opérations plus complexes sur la move_queue.

6. **La tour de contrôle a besoin de stocker pour tout ce qui lui est actuellement attribué, afin de pouvoir le libérer une fois que l'avion décolle Aircraft. TerminalCette information est actuellement enregistrée dans un std::vector<std::pair<const Aircraft*, size_t>>(size_t représentant l'indice du terminal). Cela fait que la recherche du terminal associé à un avion est effectuée en temps linéaire, par rapport au nombre total de terminaux. Cela n'est pas grave tant que ce nombre est petit, mais pour préparer l'avenir, on aimerait bien remplacer le vecteur par un conteneur qui garantira des opérations efficaces, même s'il y a beaucoup de terminaux.**

   Modifiez le code afin d'utiliser un conteneur STL plus adapté. Normalement, à la fin, la fonction find_craft_and_terminal(const Aicraft&)ne devrait plus être nécessaire.

On utilisera une map dont la clé est un Aircrafvt et la valeur est un terminal. Dans le fichier tower.cpp on remplacera la méthode find_craft_and_terminal(const Aircraft&) par std::map::find(&aircraft). La complexité sera en O(1) pour la recherche.

D- Théorie

1. **Comment a-t-on fait pour que seule la classe `Tower` puisse réserver un terminal de l'aéroport ?**

Dans la classe Tower il y a un champ privé AircraftToTerminal reserved_terminals qui liste les terminaux utilisés.

2. **En regardant le contenu de la fonction `void Aircraft::turn(Point3D direction)`, pourquoi selon-vous ne sommes-nous pas passer par une référence ?**

   Pensez-vous qu'il soit possible d'éviter la copie du `Point3D` passé en paramètre ?

Celons-moi il n’est pas nécessaire de passer par une référence car :

- Nous ne voulons pas d’effet de Bord.
- Ici sera copié le résultat direct de la création d’un Point3D dans la fonction turn et si nous voulions passer par référence le résultat sera non pas enregistré directement dans la méthode turn mais dans une variable, donc pas de réel changement concernant l’espace mémoire.

Il est possible d’éviter la copie du Point3D en créant une variable pour stocker le point puis de passer sa référence à turn.

E-Bonus

Le temps qui s'écoule dans la simulation dépend du framerate du programme.

La fonction move() n'utilise pas le vrai temps. Faites en sorte que si.

Par conséquent, lorsque vous augmentez le framerate, la simulation s'exécute plus rapidement, et si vous le diminuez, celle-ci s'exécute plus lentement.

Dans la plupart des jeux ou logiciels que vous utilisez, lorsque le framerate diminue, vous ne le ressentez quasiment pas (en tout cas, tant que celui-ci ne diminue pas trop).

Pour avoir ce type de résultat, les fonctions d'update prennent généralement en paramètre le temps qui s'est écoulé depuis la dernière frame, et l'utilise pour calculer le mouvement des entités.

Recherchez sur Internet comment obtenir le temps courant en C++ et arrangez-vous pour calculer le dt (delta time) qui s'écoule entre deux frames.

Lorsque le programme tourne bien, celui-ci devrait être quasiment égale à 1/framerate.

Cependant, si le programme se met à ramer et que la callback de glutTimerFunc est appelée en retard (oui oui, c'est possible), alors votre dt devrait être supérieur à 1/framerate.

Passez ensuite cette valeur à la fonction `move` des `DynamicObject`, et utilisez-la pour calculer les nouvelles positions de chaque avion.

Vérifiez maintenant en exécutant le programme que, lorsque augmentez le framerate du programme, vous n'augmentez pas la vitesse de la simulation.

Ajoutez ensuite deux nouveaux inputs permettant d'accélérer ou de ralentir la simulation.

Pas fait.

Task1 : Gestion des ressources

Analyse de la gestion des avions

La création des avions est aujourd'hui gérée par les fonctions `TowerSimulation::create_aircraft` et `TowerSimulation::create_random_aircraft`.

Chaque avion créé est ensuite placé dans les files `GL::display_queue` et `GL::move_queue`.

Si à un moment quelconque du programme, vous souhaitiez accéder à l'avion ayant le numéro de vol "AF1250", que devriez-vous faire ?

Si on souhaite accéder à l'avion ayant le numéro "AF1250", on devra parcourir la GL::display_queue ou bien la GL::move_queue.

Objectif 1 - Référencement des avions

A - Choisir l'architecture

Pour trouver un avion particulier dans le programme, ce serait pratique d'avoir une classe qui référence tous les avions et qui peut donc nous renvoyer celui qui nous intéresse.

Vous avez 2 choix possibles :

- créer une nouvelle classe, `AircraftManager`, qui assumera ce rôle,

- donner ce rôle à une classe existante.

Réfléchissez aux pour et contre de chacune de ces options.

Pour le restant de l'exercice, vous partirez sur le premier choix.

	Avantage	Inconvéniant
AircraftManager	Une classe = Une responsabilité Code mieux organisé et evolutif	Plus de code
Classe existante	Moins de code	Code moins organisée Plein de Class devront importer la classe contenant la méthode alors que se n'est juste pour cette méthode

B - Déterminer le propriétaire de chaque avion

Vous allez introduire une nouvelle liste de références sur les avions du programme.

Il serait donc bon de savoir qui est censé détruire les avions du programme, afin de déterminer comment vous allez pouvoir mettre à jour votre gestionnaire d'avions lorsque l'un d'entre eux disparaît.

Répondez aux questions suivantes :

1. Qui est responsable de détruire les avions du programme ? (si vous ne trouvez pas, faites/continuez la question 4 dans TASK_0)

C'est la fonction GL::timer dans le fichier opengl_interface.cpp qui a la responsabilité de détruire les avions du programme.

2. Quelles autres structures contiennent une référence sur un avion au moment où il doit être détruit ?

Les structures GL::display_queue et GL::move::queue font références aux avions au moment où ils sont détruits.

3. Comment fait-on pour supprimer la référence sur un avion qui va être détruit dans ces structures ?

Pour supprimer la référence sur un avion qui va être détruit dans ses structures on doit utiliser la méthode erase car les Aircraft sont référencer par un unique_ptr.

4. Pourquoi n'est-il pas très judicieux d'essayer d'appliquer la même chose pour votre `AircraftManager` ?

Ce n'est pas judicieux car cela nous obligerai d'avoir un AircraftManager pour qu'un Aircraft puisse exister.

Pour simplifier le problème, vous allez déplacer l'ownership des avions dans la classe `AircraftManager`.

Vous allez également faire en sorte que ce soit cette classe qui s'occupe de déplacer les avions, et non plus la fonction `timer`.

C - C'est parti !

Ajoutez un attribut `aircrafts` dans le gestionnaire d'avions.

Choisissez un type qui met bien en avant le fait que `AircraftManager` est propriétaire des avions.

std::vector<std::unique_ptr<Aircraft>> aircrafts;

Ajoutez un nouvel attribut `aircraft_manager` dans la classe `TowerSimulation`.

AircraftManager aicraft_manager;

Modifiez ensuite le code afin que `timer` passe forcément par le gestionnaire d'avions pour déplacer les avions.

On ajoute Le gestionnaire d'avions dans la move_queue et on ajoute une methode update dans la classe AircraftManager.

Faites le nécessaire pour que le gestionnaire supprime les avions après qu'ils aient décollé.

Ceci se passe dans la méthode update dans la classe AircraftManager.

Enfin, faites ce qu'il faut pour que `create_aircraft` donne l'avion qu'elle crée au gestionnaire.

Testez que le programme fonctionne toujours.

aircraft_manager.add(std::unique_ptr<Aircraft>(aircraft));

Objectif 2 - Usine à avions

A - Création d'une factory

La création des avions est faite à partir des composants suivants :

- **`create_aircraft`**
- **`create_random_aircraft`**
- **`airlines`**
- **`aircraft_types`.**

Pour éviter l'usage de variables globales, vous allez créer une classe `AircraftFactory` dont le rôle est de créer des avions.

Définissez cette classe, instanciez-la à l'endroit qui vous paraît le plus approprié, et refactorisez le code pour l'utiliser.

Vous devriez du coup pouvoir supprimer les variables globales `airlines` et `aircraft_types`.

Voir la Class AircraftFactory.

B - Conflits

Il est rare, mais possible, que deux avions soient créés avec le même numéro de vol.

Ajoutez un conteneur dans votre classe `AircraftFactory` contenant tous les numéros de vol déjà utilisés.

Faites maintenant en sorte qu'il ne soit plus possible de créer deux fois un avion avec le même numéro de vol.

Le conteneur sera std::unordered_set<std::string_view> aircraft_numbers;

On ajoute la méthode std::string new_aicraft_number() const; dans la class AircraftFactory.

C - Data-driven AircraftType (optionnel)

On aimerait pouvoir charger les paramètres des avions depuis un fichier.

Définissez un format de fichier qui contiendrait les informations de chaque `AircraftType` disponible dans le programme.

Ajoutez une fonction `AircraftFactory::LoadTypes(const MediaPath&)` permettant de charger ce fichier.

Les anciens `AircraftTypes` sont supprimés.

Modifiez ensuite le `main`, afin de permettre à l'utilisateur de passer le chemin de ce fichier via les paramètres du programme.

S'il ne le fait pas, on utilise la liste de type par défaut.

Si vous voulez de nouveaux sprites, vous pouvez en trouver sur [cette page](http://www.as-st.com/ttd/planes/planes.html)

(un peu de retouche par GIMP est necessaire)

Pas fait.

Task2 : Algorithmes

Objectif 1 - Refactorisation de l'existant

A - Structured Bindings

`TowerSimulation::display_help()` est chargé de l'affichage des touches disponibles.

Dans sa boucle, remplacez `const auto& ks_pair` par un structured binding adapté.

Voir la boucle dans TowerSimulation::display_help().

B - Algorithmes divers

1. `AircraftManager::move()` (ou bien `update()`) supprime les avions de la `move_queue` dès qu'ils sont "hors jeux".

En pratique, il y a des opportunités pour des pièges ici. Pour les éviter, `<algorithm>` met à disposition la fonction `std::remove_if`.

Remplacez votre boucle avec un appel à `std::remove_if`.

****Attention**: pour cela c'est necessaire que `AircraftManager` stocke les avion dans un `std::vector` ou `std::list` (c'est déjà le cas pour la solution filé).**

```
aircrafts.erase(std::remove_if(aircrafts.begin(), aircrafts.end(), [](auto& aircraft)

{return !aircraft->update(); }), aircrafts.end());
```

2. Pour des raisons de statistiques, on aimerait bien être capable de compter tous les avions de chaque airline.

A cette fin, rajoutez des callbacks sur les touches `0`..`7` de manière à ce que le nombre d'avions appartenant à `airlines[x]` soit affiché en appuyant sur `x`.

Dans la méthodes TowerSimulation::create_keystrokes on met la boucle :

```
for (int i = 0; i < 8; i++) {
        GL::keystrokes.emplace(i + '0', [this, i]()
                { aircraft_manager.number_aircraft_in_airline(aircraft_factory.airline(i)); });
}
```

Rendez-vous compte de quelle classe peut acquérir cette information. Utilisez la bonne fonction de `<algorithm>` pour obtenir le résultat.

```
void AircraftManager::number_aircraft_in_airline(const std::string& airline) const {
        const auto cnt = std::count_if(aircrafts.begin(), aircrafts.end(), [airline](const auto& aircraft)
                { return aircraft->get_flight_num().substr(0, 2).compare(airline) == 0; });
        std::cout << "Number of Aircraft in Airline " << airline << ": " << cnt << std::endl;
}
```

C - Relooking de Point3D

La classe `Point3D` présente beaucoup d'opportunités d'appliquer des algorithmes.

Particulairement, des formulations de type `x() = ...; y() = ...; z() = ...;` se remplacent par un seul appel à la bonne fonction de la librairie standard.

Remplacez le tableau `Point3D::values` par un `std::array` et puis,

remplacez le code des fonctions suivantes en utilisant des fonctions de `<algorithm>` / `<numeric>`:

1. `Point3D::operator*=(const float scalar)`

```
std::transform(values.begin(), values.end(), values.begin(), [scalar](const float& v) { return v * scalar; });
```

2. `Point3D::operator+=(const Point3D& other)` et `Point3D::operator-=(const Point3D& other)`

```
std::transform(values.begin(), values.end(), other.values.begin(), values.begin(),
[](const float& a, const float& b) { return a + b; });
std::transform(values.begin(), values.end(), other.values.begin(), values.begin(),
[](const float& a, const float& b) { return a - b; });
```

3. `Point3D::length() const`

```
return std::sqrt(std::reduce(values.begin(), values.end(), 0., [](float f1, float f2) { return f1 + f2 * f2; }));
```

Objectif 2 - Rupture de kérosène

Vous allez introduire la gestion de l'essence dans votre simulation.

Comme le but de ce TP est de vous apprendre à manipuler les algorithmes de la STL, avant d'écrire une boucle, demandez-vous du coup s'il n'existe pas une fonction d'`<algorithm>` ou de `<numeric>` qui permet de faire la même chose.

La notation tiendra compte de votre utilisation judicieuse de la librairie standard.

A - Consommation d'essence

Ajoutez un attribut `fuel` à `Aircraft`, et initialisez-le à la création de chaque avion avec une valeur aléatoire comprise entre `150` et `3'000`.

Décrémentez cette valeur dans `Aircraft::update` si l'avion est en vol.

Lorsque cette valeur atteint 0, affichez un message dans la console pour indiquer le crash, et faites en sorte que l'avion soit supprimé du manager.

N'hésitez pas à adapter la borne `150` - `3'000`, de manière à ce que des avions se crashent de temps en temps.

```
if (fuel-- <= 0) {

        std::cout << "Aircraft " << flight_number << " crashed." << std::endl;

        return false;

}
```

B - Un terminal s'il vous plaît

Afin de minimiser les crashs, il va falloir changer la stratégie d'assignation des terminaux aux avions.

Actuellement, chaque avion interroge la tour de contrôle pour réserver un terminal dès qu'il atteint son dernier `Waypoint`.

Si un terminal est libre, la tour lui donne le chemin pour l'atteindre, sinon, elle lui demande de tourner autour de l'aéroport.

Pour pouvoir prioriser les avions avec moins d'essence, il faudrait déjà que les avions tentent de réserver un terminal tant qu'ils n'en n'ont pas (au lieu de ne demander que lorsqu'ils ont terminé leur petit tour).

1. Introduisez une fonction `bool Aircraft::has_terminal() const` qui indique si un terminal a déjà été réservé pour l'avion (vous pouvez vous servir du type de `waypoints.back()`).

```
bool Aircraft::has_terminal() const {

        return !waypoints.empty() && waypoints.back().is_at_terminal();

}
```

2. Ajoutez une fonction `bool Aircraft::is_circling() const` qui indique si l'avion attend qu'on lui assigne un terminal pour pouvoir attérir.

```
bool Aircraft::is_circling() const {

        return !has_terminal() && !is_at_terminal && !service_done;

}
```

3. Introduisez une fonction `WaypointQueue Tower::reserve_terminal(Aircraft& aircraft)` qui essaye de réserver un `Terminal`. Si c'est possible, alors elle retourne un chemin vers ce `Terminal`, et un chemin vide autrement (vous pouvez vous inspirer / réutiliser le code de `Tower::get_instructions`).

Voir la fonction

4. Modifiez la fonction `move()` (ou bien `update()`) de `Aircraft` afin qu'elle appelle `Tower::reserve_terminal` si l'avion est en attente. Si vous ne voyez pas comment faire, vous pouvez essayer d'implémenter ces instructions :

\- si l'avion a terminé son service et sa course, alors on le supprime de l'aéroport (comme avant),

\- si l'avion attend qu'on lui assigne un terminal, on appelle `Tower::reserve_terminal` et on modifie ses `waypoints` si le terminal a effectivement pu être réservé,

\- si l'avion a terminé sa course actuelle, on appelle `Tower::get_instructions` (comme avant).

Voir la fonction

C - Minimiser les crashs

Grâce au changement précédent, dès lors qu'un terminal est libéré, il sera réservé lors du premier appel à `Aircraft::update` d'un avion recherchant un terminal.

Pour vous assurez que les terminaux seront réservés par les avions avec le moins d'essence, vous allez donc réordonner la liste des `aircrafts` avant de les mettre à jour.

Vous devrez placer au début de la liste les avions qui ont déjà réservé un terminal.

Ainsi, ils pourront libérer leurs terminaux avant que vous mettiez à jour les avions qui essayeront de les réserver.

La suite de la liste sera ordonnée selon le niveau d'essence respectif de chaque avion.

Par exemple :

```b

A - Reserved / Fuel: 100

B - NotReserved / Fuel: 50

C - NotReserved / Fuel: 300

D - NotReserved / Fuel: 150

E - Reserved / Fuel: 2500

```

pourra être réordonné en

```b

A - Reserved / Fuel: 100

E - Reserved / Fuel: 2500

B - NotReserved / Fuel: 50

D - NotReserved / Fuel: 150

C - NotReserved / Fuel: 300

```

Assurez-vous déjà que le conteneur `AircraftManager::aircrafts` soit ordonnable (`vector`, `list`, etc).

Au début de la fonction `AircraftManager::move` (ou `update`), ajoutez les instructions permettant de réordonner les `aircrafts` dans l'ordre défini ci-dessus.

Voir la 1ere instruction dans la méthode qui est un sort.

D - Réapprovisionnement

Afin de pouvoir repartir en toute sécurité, les avions avec moins de `200` unités d'essence doivent être réapprovisionnés par l'aéroport pendant qu'ils sont au terminal.

1. Ajoutez une fonction `bool Aircraft::is_low_on_fuel() const`, qui renvoie `true` si l'avion dispose de moins de `200` unités d'essence.

```
bool is_low_on_fuel() const { return fuel < 200; }
```

Modifiez le code de `Terminal` afin que les avions qui n'ont pas suffisamment d'essence restent bloqués.

```
if (in_use() && is_servicing() && !current_aircraft->is_low_on_fuel())
```

Testez votre programme pour vérifier que certains avions attendent bien indéfiniment au terminal.

Si ce n'est pas le cas, essayez de faire varier la constante `200`.

2. Dans `AircraftManager`, implémentez une fonction `get_required_fuel`, qui renvoie la somme de l'essence manquante (le plein, soit `3'000`, moins la quantité courante d'essence) pour les avions vérifiant les conditions suivantes :

\- l'avion est bientôt à court d'essence

\- l'avion n'est pas déjà reparti de l'aéroport.

Voir la fonction.

3. Ajoutez deux attributs `fuel_stock` et `ordered_fuel` dans la classe `Airport`, que vous initialiserez à 0.

Ajoutez également un attribut `next_refill_time`, aussi initialisé à 0.

```
int fuel_stock      = 0;

int ordered_fuel    = 0;

int next_refill_time = 0;
```

Enfin, faites en sorte que la classe `Airport` ait accès à votre `AircraftManager` de manière à pouvoir l'interroger.

```
AircraftManager* aircraft_manager;
```

4. Ajoutez une fonction `refill` à la classe `Aircraft`, prenant un paramètre `fuel_stock` par référence non-constante.

Cette fonction rempliera le réservoir de l'avion en soustrayant ce dont il a besoin de `fuel_stock`.

Bien entendu, `fuel_stock` ne peut pas devenir négatif.

Indiquez dans la console quel avion a été réapprovisionné ainsi que la quantité d'essence utilisée.

Voir la fonction

5. Définissez maintenant une fonction `refill_aircraft_if_needed` dans la classe `Terminal`, prenant un paramètre `fuel_stock` par référence non-constante.

Elle devra appeler la fonction `refill` sur l'avion actuellement au terminal, si celui-ci a vraiment besoin d'essence.

Voir la fonction

6. Modifiez la fonction `Airport::update`, afin de mettre-en-oeuvre les étapes suivantes.

\- Si `next_refill_time` vaut 0 :

*** `fuel_stock` est incrémenté de la valeur de `ordered_fuel`.**

*** `ordered_fuel` est recalculé en utilisant le minimum entre `AircraftManager::get_required_fuel()` et `5'000` (il s'agit du volume du camion citerne qui livre le kérosène).**

*** `next_refill_time` est réinitialisé à `100`.**

*** La quantité d'essence reçue, la quantité d'essence en stock et la nouvelle quantité d'essence commandée sont affichées dans la console.**

\- Sinon `next_refill_time` est décrémenté.

\- Chaque terminal réapprovisionne son avion s'il doit l'être.

Voir la fonction

E - Paramétrage (optionnel)

Pour le moment, tous les avions ont la même consommation d'essence (1 unité / trame) et la même taille de réservoir (`3'000`).

1. Arrangez-vous pour que ces deux valeurs soient maintenant déterminées par le type de chaque avion (`AircraftType`).

2. Pondérez la consommation réelle de l'avion par sa vitesse courante.

La consommation définie dans `AircraftType` ne s'appliquera que lorsque l'avion est à sa vitesse maximale.

3. Un avion indique qu'il a besoin d'essence lorsqu'il a moins de `200` unités.

Remplacez cette valeur pour qu'elle corresponde à la quantité consommée en 10s à vitesse maximale.

Si vous n'avez pas fait la question bonus de TASK_0, notez bien que la fonction `update` de chaque avion devrait être appelée `DEFAULT_TICKS_PER_SEC` fois par seconde.

Pas fait

Task3 :Assertions et exceptions

Objectif 1 - Crash des avions

Actuellement, quand un avion s'écrase, une exception de type `AircraftCrash` (qui est un alias de `std::runtime_error` déclaré dans `config.hpp`) est lancée.

1. Faites en sorte que le programme puisse continuer de s'exécuter après le crash d'un avion. Pour cela, remontez l'erreur jusqu'à un endroit approprié pour procéder à la suppression de cet avion (assurez-vous bien que plus personne ne référence l'avion une fois l'exception traitée). Vous afficherez également le message d'erreur de l'exception dans `cerr`.

On fait remonter l'exeception dans la méthode AircraftManager::update

2. Introduisez un compteur qui est incrémenté chaque fois qu'un avion s'écrase. Choisissez une touche du clavier qui n'a pas encore été utilisée (`m` par exemple ?) et affichez ce nombre dans la console lorsque l'utilisateur appuie dessus.

Le compteur est la valeur _nb_aircraft_crashed dans la Class AircraftManager

On l'incrémentera dans AircraftManager::update.

On ajoute la commande utilisateur M pour afficher le nombre d'avion crasher.

3. Si vous avez fini d'implémenter la gestion du kérosène (Task_2 - Objectif 2 - A), lancez une exception de type `AircraftCrash` lorsqu'un avion tombe à court d'esssence. Normalement, cette exception devrait être traitée de la même manière que lorsqu'un avion s'écrase parce qu'il a attérit trop vite.

using namespace std::string_literals;

throw AircraftCrash { flight_number + " is out of fuel" };

4. **BONUS Rédéfinissez `AircraftCrash` en tant que classe héritant de `std::runtime_error`, plutôt qu'en tant qu'alias. Arrangez-vous pour que son constructeur accepte le numéro de vol, la position, la vitesse de l'avion au moment du crash, et la raison du crash (`"out of fuel"` / `"bad landing"`). Vous utiliserez toutes ces informations pour générer le joli message de l'exception.**

Regarder la Class AircraftCrash.

On lèvera l'exception comme ceci :

throw AircraftCrash { flight_number, pos, speed, "crashed due to fuel" };

Objectif 2 - Détecter les erreurs de programmation

Pour sécuriser votre code, repassez sur les différentes fonctions de votre programme et ajoutez des assertions permettant de vérifier qu'elles sont correctement utilisées.

Voici quelques idées :

- fonctions d'initialisation appelées une seule fois

- état attendu d'un objet lorsqu'une fonction est appelée dessus

- vérification de certains paramètres de fonctions

- ...

Task4 :Templates

Objectif 1 - Devant ou derrière ?

La fonction `Aircraft::add_waypoint` permet de rajouter une étape au début ou à la fin du parcours de l'avion.

Pour distinguer ces deux cas, elle prend un argument booléen `front` (on parle alors de "flag") qui est évalué à l'exécution.

Votre objectif consistera à modifier cette fonction afin d'économiser cette évaluation.

1. Aujourd'hui, cette fonction n'est pas utilisée dans le code (oups).

Du coup, pour être sûr que vos futurs changements fonctionneront correctement, commencez par modifier le code de la classe `Aircraft` de manière à remplacer :

```cpp

waypoints = control.get_instructions(*this);

```

par

```cpp

auto front = false;

for (const auto& wp: control.get_instructions(*this)) { add_waypoint(wp, front); }

```

```
auto newWaypoints = control.reserve_terminal(*this);
if (!newWaypoints.empty()) {
	waypoints = std::move(newWaypoints);
}
```

2. Modifiez `Aircraft::add_waypoint` afin que l'évaluation du flag ait lieu à la compilation et non à l'exécution.

Que devez-vous changer dans l'appel de la fonction pour que le programme compile ?

```
template <bool front> void Aircraft::add_waypoint(const Waypoint& wp) {
	if (front) { waypoints.push_front(wp); }
	else { waypoints.push_back(wp); }
}
```

3. **BONUS En utilisant [GodBolt](https://godbolt.org/), comparez le code-assembleur généré par les fonctions suivantes:**

<table border="0">

<tr>

<td><pre lang="c++">

int minmax(const int x, const int y, const bool min) {

return x < y ? (min ? x : y) : (min ? y : x);

}

</pre></td>

<td><pre lang="c++">

template<bool min>

int minmax(const int x, const int y){

return x < y ? (min ? x : y) : (min ? y : x);

}

</pre></td>

</tr>

</table>

Les codes sont similaires mais dans la fonction avec un template il n'y a pas les conditions concernant min car on connaît sa valeur pendant la compilation contrairement a la fonction qui n'as pas de template.

Objectif 2 - Points génériques

1. Reprenez les classes dans `geometry.hpp` et inspirez-vous de `Point2D` et `Point3D` pour définir une unique classe-template `Point` paramétrée par la dimension (nombre de coordonnées) et leur type (entier/float/double).

Pour ce qui est des constructeurs, vous n'ajouterez pour le moment que le constructeur par défaut.

2. Ajoutez une fonction libre `test_generic_points` à votre programme, que vous appelerez depuis le `main`.

Placez le code suivant dans cette fonction et modifiez-le plusieurs fois, pour vérifier que le compilateur est capable de générer des classes à partir de votre template sans problème :

```cpp

Point<...> p1;

Point<...> p2;

auto p3 = p1 + p2;

p1 += p2;

p1 *= 3; // ou 3.f, ou 3.0 en fonction du type de Point

```

Voir la fonction static dans la class Point.

3. Ajoutez le constructeur à 2 paramètres de `Point2D` et le constructeur à 3 paramètres de `Point3D` dans votre classe-template.

Modifiez `Point2D` et `Point3D` afin d'en faire des alias sur des classes générées à partir du template `Point` (respectivement, 2 floats et 3 floats).

Point(Type x, Type y) : values { x, y }

Point(Type x, Type y, Type z) : values { x, y, z } {}

Vérifiez que votre programme compile et fonctionne comme avant.

4. Dans la fonction `test_generic_points`, essayez d'instancier un `Point2D` avec 3 arguments.

Que se passe-t-il ?

Il y a une erreur durant l'exécution.

Comment pourriez-vous expliquer que cette erreur ne se produise que maintenant ?

Durant la compilation on ne fait pas attention au nombre d'argument.

5. Que se passe-t-il maintenant si vous essayez d'instancier un `Point3D` avec 2 arguments ?

On aura un point avec x et y qui auront comme valeur respectivement le 1er et le 2eme argument, z aura une valeur aléatoire.

Utilisez un `static_assert` afin de vous assurez que personne ne puisse initialiser un `Point3D` avec seulement deux éléments.

Faites en de même dans les fonctions `y()` et `z()`, pour vérifier que l'on ne puisse pas les appeler sur des `Point` qui n'ont pas la dimension minimale requise.

6. Plutôt qu'avoir un constructeur pour chaque cas possible (d'ailleurs, vous n'avez pas traité tous les cas possibles, juste 2D et 3D), vous allez utiliser un variadic-template et du perfect-forwarding pour transférer n'importe quel nombre d'arguments de n'importe quel type directement au constructeur de `values`.

Vous conserverez bien entendu le `static_assert` pour vérifier que le nombre d'arguments passés correspond bien à la dimension du `Point`.

En faisant ça, vous aurez peut-être désormais des problèmes avec la copie des `Point`.

Que pouvez-vous faire pour supprimer l'ambiguité ?

7. **BONUS En utilisant SFINAE, faites en sorte que le template `Point` ne puisse être instancié qu'avec des types [arithmétiques](https://en.cppreference.com/w/cpp/types/is_arithmetic).**

template <typename... Arg, std::enable_if_t<std::is_arithmetic_v<TypePoint>, bool> = true>

Architecture

Voir les réponses aux questions dans la rubrique Réponses qui explique les choix d'implémentation.
De plus dans le code il y a quelques commentaires.

Voici les touches clavier et leurs actions :

Touche	Action
X	Quitter le programme
Q	Quitter le programme
C	Ajouter un Aircraft aléatoire
+	Zoom
-	DeZoom
F	Plein écran / Retour en fenêtre
U	Augmente la fréquence d'image
D	Diminue la fréquence d'image
P	Pause / Start
M	Affiche le nombre d'avion crashée
0-7	Nombre d'avions a l'écran de la compagnie aérienne X.

Les questions obligatoires ont été traiter ainsi que les bonus de la Task3 et de la Task4 Objectif 1 et 2.

Difficultés

Bug :

- De temps en temps lorsque l'on rajoute de l'essence a un avion on ajoute plusieurs fois 0 essence dans l'avion, puis au dernier cycle on ajoute le nombre d'essence qu'il faut
- Il arrive extrêmement rarement que dans Terminal::start_service il y ai un l'assert suivant qui s'active :
  assert(aircraft.distance_to(pos) < DISTANCE_THRESHOLD);

Ressentis

J'ai bien aimé le fait qu'il y a un code de départ car on peut se focaliser directement sur le code sans passer par les étapes préliminaire (makefile, dossier/fichier de base…)

C'est la 1ère fois que l'on a un code de base pour un projet, donc cela était original dans un premier temps de comprendre le code. En entreprise on sera aussi confronté à ce type de cas donc c'est bien d'avoir déjà une certaine expérience dans la compréhension d'un code écris par un autre et de l'évoluer.

Apprentissage

C'est le 1er projet où nous devons partir d'un code déjà existant et l'enrichir.
Cela m'a appris à comprendre un code déjà existant et à l'enrichir sans repartir de 0 tout en respectant sa sémantique.
De plus cela m'a permis de consolidé mes acquis sur la programmation en C++ et la Programmation Orientée Objet.